Master Thesis

The Seat Assignment Problem for Aeroplane Boarding

Jens Doveren

First examiner: Priv.-Doz. Dr. Walter Unger
Second examiner: Prof. Dr. Marco Lübbecke

Contents

Chapter 1

Introducion

In this thesis, we investigate possible optimisations of the aeroplane boarding process. To accomplish this, we define and mathematically model the process, formulate it as a minimisation problem, and propose a solution procedure. We show that the problem is computationally hard in a theoretical sense and investigate the quality of heuristic approaches as well as some of its online properties.

The boarding process is of particular interest for airlines to improve profitability as well as customer satisfaction. Since boarding policy changes pose a lower implementation barrier than buying new planes or making changes to existing infrastructure, advances in aeroplane boarding research have the potential to positively impact customers and airlines relatively immediately.

1.1 Literature Survey

One effect of long passenger boarding times are flight delays, which are an ever-present problem in air traffic. According to [Sch10], as many as 35.5 percent of European flights were delayed by more than 15 minutes in 2007. Delays are by no means merely a nuisance - according to a study of the economic impact of flight delays in the United States by [Bal+10], delays in 2007 incurred cost of US$ 31.2 bn, of which US$ 16.7 bn were borne by passengers in the form of missed connections and working hours. In addition to reducing delays, speeding up the boarding process can help reduce the time a plane spends at the gate, which has direct financial implications for the airline. It has been estimated by [CTA04] that reducing the strategic time buffer of an Airbus A320, one of the most widely used passenger jets in the industry, by one minute saves the airline between €11 and €48.

Due to the potential economic benefits, research into optimal boarding procedures has been conducted for some time. A general introduction into the topic as well as an overview of many of the strategies explored so far can be found in [JN15]. It is important to note that this overview paper, like the larger part of the body of research, only considers procedures where passengers are assigned their seats prior to boarding, as opposed to *open seating*, where passengers choose their seats during boarding. This differentiates the scenario explored in this here thesis from those most commonly studied - we aim to understand the problem of assigning passengers their seat upon entering the plane, while the order in which they arrive is out of our control.

In the context of the traditional scenario in which passengers are assigned their seats prior to boarding and the sequence in which they board may be altered, [JN15] cites STEFFEN*'s method* as the fastest procedure to board by seat. Initially proposed in [Ste08], it results from generalising a seating pattern that was the result of a MARKOV *chain Monte Carlo* optimisation. STEFFEN's method primarily focusses on the time passengers spend stowing their luggage in overhead compartments and garnered positive reception in both academic as well as mainstream publications. The fact that the development of a boarding procedure gets featured in mainstream media outlets such as [Mou11], [Str14] and [Sto14] points to significant interest in the topic even by the general public. The development of efficient boarding procedures may also be directly commissioned by airlines, as was the case for [Bri+05], which was financed and tested at AMERICA WEST AIRLINES in 2003. By implementing the *reverse pyramid* boarding scheme suggested in the paper, *America West Airlines* reportedly reduced their boarding times by 20 percent. The boarding methods mentioned here will be properly defined in chapter 5 and examined in terms of their performance in chapter 6.

The methods to model, simulate and optimise the boarding process vary widely from *multi-agent based* simulation using varying boarding policy compliance rates where passengers move at individually different speeds in [AVB09] to using a *cell-based* simulation where both walking and stowing speeds are chosen from a range in the comparative study [JM17]. Since we will use *mixed integer programming (MIP)* in this thesis, it shall be noted that this approach to boarding optimisation was first explored in [Baz07] to assign passengers who already know their seats to boarding groups. Additional work using this technique was done in [MS16], where the authors used *mixed integer programming* to assign passengers to seats based on the number of carry-on luggage items, while assuming a constant walking speed and infinite overhead compartment size. Using this optimisation as a building block, the same authors suggested a three stage process to yield more stable seat assignments in [MSK18]. In this process, their previously developed *MIP* assigns seats to passengers based on their carry-on items, then a second *MIP* aims to stabilise the assignment without affecting the boarding time and as a final step, the passenger boarding sequence is computed using STEFFEN*'s method*.

STEFFEN*'s method* also serves as the boarding sequence in the recently published [SMK19], in which the authors model moving through the plane using a MIP in order to assign passengers seats. It is assumed that moving speed is constant and independent of the individual passenger and the stowing speed is determined by the number of carry-on items, as is the case in [MSK18].

Recently, [WT19] has investigated the use of mixed integer programming to optimise boarding sequences, while imposing rules for passenger behaviour inside the plane similar to the ones used in this thesis. Since we shall use the same passenger behaviour regime, this work is essentially a variation of the aforementioned paper in which we investigate the optimisation possibilities of assigning seats rather than modifying the boarding sequence.

1.2 Own Contributions

In section 2, this thesis introduces and formalises a boarding scenario in which passengers are assigned their seats upon entering the plane, while the order of the passengers cannot be changed. We call this scenario the *Boarding an Aeroplane Problem (BAP)*. Such a scenario is different from - and in a certain sense sits in between - the two classical approaches of open seating and assigning seats before the boarding starts. The motivation to study this scenario is twofold - on one hand it is interesting from a purely theoretical point of view to explore how the properties of the aeroplane boarding problem change when instead of effectively rearranging the queue given the seat assignments we are allowed to assign seats but not change queueing order. On the other hand, practical implementations of a boarding policy which allows passengers to first queue and then assign them seats upon entering become feasible when using electronic ticketing systems and smart devices that tell passengers their assigned seats. Avoiding rearranging passengers or forming boarding groups this way has the potential to have lower requirements in terms of personnel and infrastructure, while potential gains in boarding speed are explored in this thesis. It is important to note that we only consider perfectly rectangular seat layouts with a single aisle in the middle, as it is commonly found in short and medium-haul flights.

In section 3, we show that BAP is NP-hard by reduction of *3-Partition.* Since it is hence unlikely that there can be an efficient algorithm for solving BAP, unless $P = NP$, this finding motivates the study of heuristics and approximations. The NP-hardness proof may also be of interest to readers with a background in machine scheduling, as it is easy to imagine BAP as such a problem, but we have been unable to find any literature about this specific setting.

In section 4, we present two mixed integer programming approaches to solve BAP exactly. One of these MIPs is a compact formulation while the other utilises a larger number of lazy constraints. Their computational performance is compared in section 6.1.

In section 5, we formally define various heuristic approaches to the problem. Some of these were created by ourselves, while others were taken from existing literature, such as STEFFEN*'s method* and the *reverse pyramid* scheme. We characterise a set of problem instances for which one of our heuristics is optimal in theorem 2.

In section 6, we present a computational comparing the different approaches on a set of problem instances in terms of their computation time, solution quality and robustness when compared to the exact MIP solution. We shall see that in many cases, heuristic solutions are of the same quality as the ones produced by our exact approaches.

In section 7 we present a number of potentially interesting problems surrounding *BAP* that were out of scope for this thesis. These focus primarily around the online properties of the problem, which are of particular interest, since they directly relate to real-world scenarios. While all previous sections assume that when computing a seat assignment one has knowledge of the entire passenger sequence, in section 7.2 we consider scenarios where that knowledge is limited. Such a scenario might be that the passenger sequence only becomes known during the seat assignment process, and every passenger must be assigned a seat before the next one is presented. In reality, such a situation arises when passengers are not forced into a proper queue but rather wait to enter the plane in a drove. While they still enter the plane in a given order, the seat assigner cannot look ahead past the passenger at the plane's entrance. We show that all approaches that ignore the distribution of walking and stowing speeds over all passengers, which includes STEFFEN*'s method* and the *reverse pyramid scheme*, produce results that are arbitrarily worse than the optimal seat assignment.

Chapter 2

Problem Formulation

We consider the problem of assigning a queue of passengers seats in an aeroplane upon entering, called the *Boarding an Aeroplane Problem*, or *BAP* for short.

Definition 1 (Boarding an Aeroplane Problem)**.** An instance of *BAP* consists of the following:

- a finite ordered set $\mathfrak{P} = (p_1, \ldots)$ of passengers.
- a finite ordered set $\mathfrak{R} = (r_1, \ldots)$ of rows.
- numbers of seats per row $k_1, k_2 \in \mathbb{Z}_{\geq 0}$ for each side of the plane. We define $k \coloneqq k_1 + k_2$ as the total number of seats per row.
- a finite set $\mathfrak{S} \coloneqq \biguplus_{r \in \mathfrak{R}} (\mathfrak{S}_r^1 \uplus \mathfrak{S}_r^2)$ of seats, where $\mathfrak{S}_r^1 \coloneqq ((r, 1), \ldots, (r, k_1))$ and $\mathfrak{S}_r^2 \coloneqq ((r, k_1 + 1), \ldots, (r, k_1 + k_2))$ are the seats in row $r \in \mathfrak{R}$ on each side of the plane. It holds that $|\mathfrak{P}| \leq |\mathfrak{S}|$. To refer to the row a seat $s \in \mathfrak{S}$ is in, we write $r(s)$.
- for each row $r \in \mathfrak{R}$, for each passenger $p \in \mathfrak{P}$ the time the passenger p takes to pass through the aisle section by row r is given by $t^w_{p,r} \in \mathbb{Q}_{\geq 0}$.
- for each passenger $p \in \mathfrak{P}$, the time the passenger p occupies the aisle when stowing away their luggage to take a seat in row r is given by $t^s_{p,r} \in \mathbb{Q}_{\geq 0}$.

For readers with a background in mixed integer programming, the details of the problem are probably most easily understood by looking at the MIP formulation, which can be found in section 4. Since this thesis presents the first formulation of the problem, we invest the effort to define the problem in a formal manner.

For every passenger $p \in \mathfrak{P}$ and time point $t \in \mathbb{Q}_{\geq 0}$, the position of p is given by $\lambda : \mathfrak{P} \times \mathbb{Q}_{\geq 0} \to \{q\} \uplus \mathfrak{R} \uplus \mathfrak{S}$. It can be the queueing position q, a row or a seat.

The goal is to compute an injective seat assignment $\sigma : \mathfrak{P} \to \mathfrak{S} : p \mapsto ((r(p), s(p))$ such that it minimises the total boarding time $T(\sigma)$, which is defined as follows:

$$T : \mathfrak{S}^{\mathfrak{P}} \to \mathbb{Q}_{\geq 0} : \min_{t \in \mathbb{Q}_{\geq 0}} \{t \mid \forall p \in \mathfrak{P} : \lambda(p, t) = \sigma(p)\}$$

Since we assign passengers their seats upon entering, situations where passengers have to work their way past an already seated passenger within a row can always be avoided by boarding *window first*. Hence we are often more interested in the row that a passenger is assigned to, which we denote as follows:

$$\rho : \mathfrak{P} \to \mathfrak{R} : p \mapsto r(\sigma(p))$$

Crucially for the minimisation process, the passenger position at any given point of time is uniquely defined by the following constraints:

- Every passenger starts in the queue:

$$\forall p \in \mathfrak{P} : \lambda(p, 0) = q$$

- Passengers enter the aeroplane in queueing order:

$$\forall p \in \mathfrak{P}\ \forall p' \in \mathfrak{P}_{\geq p} : (\lambda(p, t) = q \implies \lambda(p', t) = q)$$

- At any given point of time, every position that is not the queue is occupied by at most one passenger.

$$\forall p, p' \in \mathfrak{P}\ \forall t \in \mathbb{Q}_{\geq 0} : (\lambda(p, t) = \lambda(p', t) \implies p = p' \vee \lambda(p, t) = q)$$

- Passengers do not walk backwards. Once they have left the queue, they do not enter it again. They do not walk back into past rows and do not leave their seat:

$$\forall p \in \mathfrak{P}\ \forall r \in \mathfrak{R}\ \forall r' \in \mathfrak{R}_{\leq r}\ \forall t \in \mathbb{Q}\ \forall t' \in \mathbb{Q}_{\geq t} :$$
$$\begin{cases} \lambda(p, t) \neq q & \implies \lambda(p, t') \neq q \\ \lambda(p, t) = r & \implies \lambda(p, t') \neq r' \\ \lambda(p, t) = \sigma(p) & \implies \lambda(p, t') = \sigma(p) \end{cases}$$

- Passengers move on to the next position as early as possible:

$$\forall p \in \mathfrak{P}\ \forall t \in \mathbb{Q}_{\geq 0}\ \forall i \in [|\mathfrak{R}|]\ \ \forall t' \in \mathbb{Q}_{\geq t + t^w_{p, r_i}} :$$
$$\lambda(p, t) = r_i \implies \begin{cases} \lambda(p, t') = \sigma(p) & \vee \\ \exists\ p' \in \mathfrak{P}_{<p} : \lambda(p', t') = r_{i+1} & \vee \\ \exists\ r' \in \mathfrak{R}_{>r_i} : \lambda(p, t') = r' \end{cases}$$

- Passengers do not move faster than their movement speed allows. After having stowed their luggage, passengers take their seat immediately.

$$\forall\ p \in \mathfrak{P}\ \forall r \in \mathfrak{R}_{<\rho(p)} :$$
$$\sup\{t \in \mathbb{Q}_{\geq 0} \mid \lambda(p,t) = r\} - \inf\{t \in \mathbb{Q}_{\geq 0} \mid \lambda(p,t) = r\} \geq t^w_{p,r}$$
$$\forall\ p \in \mathfrak{P} :$$
$$\sup\{t \in \mathbb{Q}_{\geq 0} \mid \lambda(p,t) = \rho(p)\} - \inf\{t \in \mathbb{Q}_{\geq 0} \mid \lambda(p,t) = \rho(p)\} = t^s_{p,\rho(p)}$$

- Passengers cannot teleport, that is they must move from the queue to the first row, from each row to the following row and from their row to their seat:

$$\begin{aligned}
&\forall p \in \mathfrak{P}\ \forall t \in \mathbb{Q}_{\geq 0}\ \forall i \in [|\mathfrak{R}|] : \\
\lambda(p,t) = r_1 &\implies \exists t' \in \mathbb{Q} \cap [0,t) : \lambda(p,t') = q \\
\lambda(p,t) = r_i &\implies \exists t' \in \mathbb{Q} \cap [0,t) : \lambda(p,t') = r_{i-1} \\
\lambda(p,t) = \sigma(p) &\implies \exists t' \in \mathbb{Q} \cap [0,t) : \lambda(p,t') = r(p)
\end{aligned}$$

Chapter 3

NP-Hardness

In this chapter we show that the aeroplane boarding problem as defined in 1 is strongly NP-hard. This will be done via reduction of the *3-Partition* problem, which is known to be strongly NP-hard (see [GJ75]).

Definition 2 (3-Partition)**.** An instance I of *3-Partition* is defined by the integers $B, m \in \mathbb{N}_{>0}, A := (a_1, \dots, a_{3m}) \in \mathbb{Z}^{3m}_{>0}$, such that $\frac{B}{4} < a_i < \frac{B}{2}$ for all $i \in [3m]$ and $\sum_{i=1}^{3m} a_i = mB$.

The decision problem is whether there exists a series of multisets $(A_1, \dots A_m)$ such that $A_j \subseteq A$ and $\sum_{a \in A_j} a = B$ for all $j \in [m]$ and for each $i \in [3m]$ there exists exactly one $j \in [m]$ such that $a_i \in A_j$. For instances for which these conditions hold, we write $I \in$ *3-Partition*.

Theorem 1. *BAP* is strongly NP-hard.

Proof. Consider an instance I of *3-Partition*, which is defined as follows:

- $m, B \in \mathbb{Z}_{>0}$
- $A := (a_1, \dots, a_{3m}) \in \mathbb{Z}^{3m}_{>0} \ : \ \frac{B}{4} < a_i < \frac{B}{2} \ \forall i \in [3m], \sum_{i=1}^{3m} a_i = mB$

We implicitly define a transformation f from instances of *3-Partition* to instances of *BAP* by defining $f(I)$ as follows:

- $\mathfrak{P} := [5m]$
- $\mathfrak{R} := [2m^2 - m]$
- $k := 5$
- $t^w_{p,r} := 0 \ \forall p \in \mathfrak{P} \ \forall r \in \mathfrak{R}$

For convenience, we introduce the following notation for *indicator rows*, that is the rows in the plane from which we will be able to read a 3-partition of the a_i, should one exist:

$$\mathbb{I} := \left\{ \mathbb{I}_j := j + \sum_{i=1}^{j-1} 4(m-i) \;\middle|\; j \in [m] \right\}$$

The passengers will be partitioned into three sets that serve distinct functions in the reduction, namely the two sets of *synchronisation passengers* $\mathbb{S}_1$ and $\mathbb{S}_2$ as well as the *partition passengers* $\mathbb{P}$. These are defined as follows:

$$\mathbb{S}_1 := [m]\,,\ \mathbb{P} := \{m+1, \ldots, 4m\},\ \mathbb{S}_2 := \{4m+1, \ldots, 5m\}$$

Using this notation, we define the stowing times for each passenger $p \in \mathfrak{P}$ and $r \in \mathfrak{R}$ as follows:

$$t^s_{p,r} := \begin{cases} (m-p)B & \text{if } p \in \mathbb{S}_1 \text{ and } r = \mathbb{I}_{m+1-p} \\ a_{p-m} & \text{if } p \in \mathbb{P} \text{ and } r \in \mathbb{I} \\ (p-(4m+1))B & \text{if } p \in \mathbb{S}_2 \text{ and } r = \mathbb{I}_{5m-p+1} \\ (mB)^2 & \text{otherwise} \end{cases}$$

An optimal assignment on such an instance can be seen in figure (3.1). In order to proof the correctness of this reduction, we first proof the following lemma:

Lemma 1. *Let I be an instance of 3-Partition with notation as defined in (2). It holds that* $\text{cost}(\text{opt}(f(I))) \geq mB$.

Proof. Let I be an instance of *3-Partition*. Consider an optimal seat assignment σ for the *BAP* instance $f(I)$. Note that if $\sigma(\hat{p}) \notin \mathbb{I}$ for any $\hat{p} \in \mathfrak{P}$ we have $\text{cost}(\sigma) \geq (mB)^2 \geq mB$. In order to see the same for the remaining case where $\sigma(p) \in \mathbb{I}$ for all $p \in \mathfrak{P}$ we consider the minimum stowing time μ_p for any passenger regardless of their seat:

$$\mu_p := \min_{r \in \mathfrak{R}} t^s_{p,r}$$

This yields the accumulated stowing time acc^s for all passengers as follows:

$$\begin{aligned}
\mathrm{acc}^s &\coloneqq \sum_{p \in \mathfrak{P}} \mu_p \\
&= B \sum_{p \in \mathbb{S}_1} (m-p) + B \sum_{p \in \mathbb{P}} a_{p-m} + B \sum_{p \in \mathbb{S}_2} (p - (4m+1)) \\
&= B \left(\frac{m(m-1)}{2} \right) + mB + B \left(\frac{m(m-1)}{2} \right) \\
&= m^2 B
\end{aligned}$$

Since $\sigma(p) \in \mathbb{I}$ for all $p \in \mathfrak{P}$ it holds that:

$$\mathrm{cost}(\sigma) \geq \frac{\mathrm{acc}^s}{|\mathbb{I}|} = \frac{m^2 B}{m} = mB$$

□

We show that any positive instance I of *3-Partition* satisfies $\mathrm{cost}(\mathrm{opt}(I)) = mB$. Let I be a positive instance of *3-Partition*, that is one for which there exists a partition $(A_1, \ldots, A_m)$ fulfilling the requirements defined in (2). Since mB is a lower bound on the makespan induced by any seat assignment, as shown in (1), any assignment σ with $\mathrm{cost}(\sigma) = mB$ is optimal. One such optimal assignment σ is given as follows:

$$\sigma(p) \coloneqq \begin{cases} \mathbb{I}_{m+1-p} & \text{if } p \in \mathbb{S}_1 \\ \mathbb{I}_j & \text{if } p \in \mathbb{P} \text{ and } a_{p-m} \in A_j \\ \mathbb{I}_{5m+1-p} & \text{if } p \in \mathbb{S}_2 \end{cases}$$

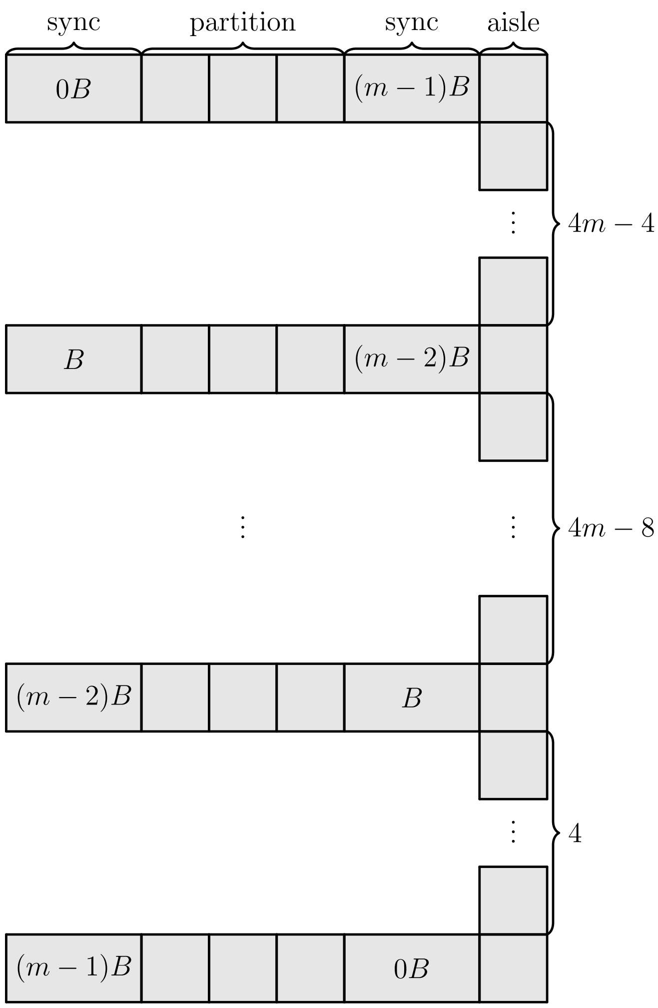


Figure 3.1: An optimal assignment in a BAP instance generated in reduction.

In order see that σ results in a boarding schedule with a makespan of mB, it is advised to look at figure 3.1 and simulate the boarding process in m phases of length B.

During the first phase, all passengers from $\mathbb{S}_1$ enter the plane and all indicator rows except $\mathbb{I}_1$ are blocked by a stowing passenger from $\mathbb{S}_1$. All passengers from $\mathbb{P}$ enter the plane and those $p \in \mathbb{P}$ for which $\sigma(p) = \mathbb{I}_1$ take their seat. This process takes exactly B seconds since this is the accumulated stowing time of these passengers by construction of $f(I)$ and σ. Since there is aisle space for $4m-4$ passengers between $\mathbb{I}_1$ and the obstruction by the passenger stowing in $\mathbb{I}_2$, passengers who are not assigned a seat in $\mathbb{I}_1$ do not obstruct those who are.

The last passengers to enter are the synchronisation passengers from $\mathbb{S}_2$, for all of whom there is space in the aisle between $\mathbb{I}_1$ and $\mathbb{I}_2$, except for the final one p_{5m}, for whom $\sigma(p_5) = \mathbb{I}_1$ holds and who will finish stowing at mB.

During the second phase, that is after B time has passed, the synchronisation passenger in $\mathbb{I}_2$ has finished and the remaining passengers behave in a similar fashion as they did in the first phase. The aisle space between $\mathbb{I}_1$ and $\mathbb{I}_2$ is four rows shorter to account for the four passengers from $\mathbb{P}$ and $\mathbb{S}_2$ seated in $\mathbb{I}_1$. The stowing time for the synchronisation passenger from $\mathbb{S}_2$ in $\mathbb{I}_2$ is shortened by B to account for the later start of the phase so they will finish stowing at mB.

All following phases follow the same pattern, all having the synchronisation passenger from $\mathbb{S}_2$ finish stowing at mB. Hence it holds that $\text{cost}(\sigma) = mB$, proofing that $\text{cost}(\text{opt}(f(I))) = mB$ for any positive *3-Partition* instance I.

Conversely we show that for any negative instance I of *3-Partition* it holds that $\text{cost}(\text{opt}(I)) > mB$. Let I be a negative instance of *3-Partition* as defined in 2, that is one for which for all partitions $(A_1, \ldots, A_m)$ there exists a $j \in [m]$ such that $\sum_{a \in A_j} a > B$. Consider the *BAP* instance $f(I)$ and assume that there exists an optimal seat assignment σ such that $\text{cost}(\sigma) = mB$.

By construction of the stowing times, only the indicator rows $\mathbb{I}$ can be used to achieve that makespan. Also by construction, exactly one passenger from each of the synchronisation sets $\mathbb{S}_1$ and $\mathbb{S}_2$ is seated in each indicator row and their respective stowing times add up to $(m-1)B$. For the overall makespan of σ to be no larger than mB, the remaining stowing time budget for each indicator row is B. Since there are as many remaining indicator row seats as there are passengers in $\mathbb{P}$, all three remaining seats per indicator row must be utilised. This implies a partition $(A_1, \ldots, A_m)$ on $\mathbb{P}$ and the associated A from the original *3-Partition* instance. Since there exists a $j \in [m]$ such that $\sum_{a \in A_j} a > B$ by assumption, there must be an indicator row that has an accumulated stowing time larger than mB using the assignment σ. This is contradicts $\text{cost}(\sigma) = mB$. Hence such an assignment cannot exist and since mB is a lower bound on the makespan as per lemma (1) any negative instance I of *3-Partition* must imply an optimal makespan of $f(I)$ larger than mB. □

Chapter 4

MIP Formulation

In a mixed integer programming setting we optimise boarding time by minimising a makespan variable $C_{\max}$ that is required to be greater or equal to the individual time that every passenger is seated. What makes such a formulation of *BAP* non-trivial is the fact that on one hand, we need to model arrival times for passengers at every row to be able to enforce minimum stay durations (i. e. walking and stowing times), and on the other hand the seat assignment is a variable. In combination, those two factors mean that at modelling time, for a given passenger, we do not know the specific row from which leaving means having taken a seat.

4.1 Standard Formulation

The solution we present here uses the *big M* method to deactivate certain constraints for passengers that have already taken their seat to effectively allow them to pass through other passengers, to no longer obstruct other passengers and to rush to the end of the plane in zero time.

In order to solve a given instance of *BAP*, with notation as defined in 1, we solve the following mixed integer program, where we define $\mathfrak{R}^* := \mathfrak{R} \uplus \{|\mathfrak{R}| + 1\}$:

$$
\begin{array}{llllr}
\min & C_{\max} & & & \\
\text{s. t.} & C_{\max} & \geq t^{\text{arr}}_{p,|\mathfrak{R}|+1} & \forall p \in \mathfrak{P} & (4.1)\\
& \sum_{r \in \mathfrak{R}} x_{p,r} = 1 & & \forall p \in \mathfrak{P} & (4.2)\\
& \sum_{p \in \mathfrak{P}} x_{p,r} \leq k & & \forall r \in \mathfrak{R} & (4.3)\\
& t^{\text{arr}}_{p,r+1} & \geq t^{\text{arr}}_{p,r} + t^{s}_{p,r} x_{p,r} + t^{w}_{p,r} \sum_{r'=r+1}^{|\mathfrak{R}|} x_{p,r'} & \forall p \in \mathfrak{P} \quad \forall r \in \mathfrak{R} & (4.4)\\
& t^{\text{arr}}_{p,r} & \geq t^{\text{arr}}_{p',r+1} - M(1 - \sum_{r'=r+1}^{|\mathfrak{R}|} x_{p,r'}) & \forall p \in \mathfrak{P} \quad \forall p' \in \mathfrak{P}_{<p} \quad \forall r \in \mathfrak{R} & (4.5)\\
& t^{\text{arr}}_{p,r+1} & \geq x_{p,r}(\sum_{r'=1}^{r-1} t^{w}_{p,r'} + t^{s}_{p,r}) & \forall p \in \mathfrak{P} \quad \forall r \in \mathfrak{R} & (4.6)\\
& x_{p,r} & \in \{0,1\} & \forall p \in \mathfrak{P} \forall r \in \mathfrak{R} & (4.7)\\
& t^{\text{arr}}_{p,r} & \in \mathbb{Q}_{\geq 0} & \forall p \in \mathfrak{P} \forall r \in \mathfrak{R}^* & (4.8)\\
& C_{\max} & \in \mathbb{Q}_{\geq 0} & & (4.9)
\end{array}
$$

The solution of the *BAP* instance is a seat assignment σ, which is encoded in binary decision variables $x_{p,r}$, defined in (4.7) for every passenger $p \in \mathfrak{P}$ and row $r \in \mathfrak{R}$. The makespan variable $C_{\max}$ in (4.9) is chosen to be continuous although it is guaranteed to be integral by the definition of *BAP*. The reasoning behind this is to not needlessly restrict the solver and to avoid branching on variables that are not decision variables. For the same reason, the variables $t^{\text{arr}}_{p,r}$ defined in (4.8), encoding the time that a given passenger $p \in \mathfrak{P}$ arrives at a row $r \in \mathfrak{R}^*$, are continuous as well. Since arriving in one row is considered to be the same as leaving the previous one, we add a virtual row $|\mathfrak{R}|+1$ after the last row to indicate leaving the last row and call the set of rows including this virtual row $\mathfrak{R}^*$.

To ensure that the valuation of the decision variables $x_{p,r}$ encodes a valid seat assignment, every passenger $p \in \mathfrak{P}$ must be assigned exactly one row $r \in \mathfrak{R}$, which is required in (4.2). Since we assign passengers to rows rather than seats we need to make sure that we cannot assign more passengers to a row than the plane has seats per row, which is enforced in (4.3). The reason that we assign to rows rather than seats is that in our definition of *BAP* there is no difference in stowing time for seats within the same row, as well as no penalty for moving past seated passengers in a row. Assigning to seats under these conditions introduces symmetry into the model, since all passengers in a row can be permuted without affecting the makespan. Since symmetry is computationally disadvantageous in mixed integer programs we chose to assign to rows and interpret the result as a seat assignment from the window to aisle.

The makespan condition (4.1) defines boarding as complete once all passengers have left the last row. This definition requires a loosening of constraints once a passenger has taken their seat such that their virtual way to the end of the plane does not interfere with actual passengers walking or stowing. The remaining constraints concern the relation between arrival and departure times of passengers in rows.

A passenger $p \in \mathfrak{P}$ can only leave a row $r \in \mathfrak{R}$ after their arrival. Should p be seated in r they must spend stowing time $t^s_{p,r}$, should they be seated in a row behind r, that is some $r' \in \mathfrak{R}_{>r}$ they must spend walking time $t^w_{p,r}$. All of these constraints are encoded in (4.4).

Row usage is exclusive, i. e. passengers block other passengers from entering the row they are occupying, hence each passenger can only enter a given row once all previous passengers have left that row. This condition uses the order on the passenger set $\mathfrak{P}$ and is encoded in (4.5). To avoid passengers that have already taken their seat obstructing other passengers, they are allowed to enter occupied rows once they have taken their seat, which is encoded as a *big M* condition. The M can be chosen as the maximum amount of time that a passenger might spend in any row.

In an attempt to improve the dual bound during the branch-and-bound process, we add a lower bound on the arrival times of any given combination of passenger and row in (4.6). This bound is easily computed as the sum of walking times for all rows before the given one and the stowing time for the given row. The effect that adding these constraints has on the solution process will be evaluated empirically in section 6.1.

4.2 Alternative Formulation

One weakness of the previous model is that since passengers only get assigned to seats during the optimisation, it is unknown where a passenger sits at modelling time. As a consequence, passengers must walk to the end of the plane after having taken their seat as ghosts.

The desire to overcome this weakness motivates the alternative MIP formulation presented here. Instead of passengers, we imagine seats moving to their predetermined position in the plane, which has the advantage that we know at modelling time where a seat will go and can hence hard-code where the assigned passenger will stow their luggage. The obvious downside of such a formulation is that the order of the passengers given in the instance must be transferred onto the seats, which have no inherent order. This is accomplished using a *big M* condition, again resulting in computational difficulties.

$$
\begin{aligned}
\min \quad & C_{\max} & & & \\
\text{s. t.} \quad & C_{\max} & \geq t^{\text{leave}}_{r,(r,s)} & \quad \forall (r,s) \in \mathfrak{S} & (4.10)\\
& \sum_{(r,s)\in\mathfrak{S}} x^p_{(r,s)} & = 1 & \quad \forall p \in \mathfrak{P} & (4.11)\\
& \sum_{p\in\mathfrak{P}} x^p_{(r,s)} & \leq 1 & \quad \forall (r,s) \in \mathfrak{S} & (4.12)\\
& t^{\text{arr}}_{q,(r,s)} & = t^{\text{leave}}_{q-1,(r,s)} & \quad \forall (r,s) \in \mathfrak{S}\ \forall q \in \mathfrak{R}_{1<q\leq r} & (4.13)\\
& t^{\text{leave}}_{q,(r,s)} & \geq t^{\text{arr}}_{q,(r,s)} + \sum_{p\in\mathfrak{P}} x^p_{(r,s)} t^w_{p,r} & \quad \forall (r,s) \in \mathfrak{S}\ \forall q \in \mathfrak{R}_{<r} & (4.14)\\
& t^{\text{leave}}_{r,(r,s)} & = t^{\text{arr}}_{r,(r,s)} + \sum_{p\in\mathfrak{P}} x^p_{(r,s)} t^s_{p,r} & \quad \forall (r,s) \in \mathfrak{S} & (4.15)\\
& t^{\text{arr}}_{q,(r,s)} & \geq t^{\text{leave}}_{q,(r',s')} - M\left(2 - x^p_{(r,s)} - \sum_{p'\in\mathfrak{P}_{<p}} x^{p'}_{(r',s')}\right) & \quad \forall (r,s) \in \mathfrak{S}\ \forall (r',s') \in \mathfrak{S}\ \forall p \in \mathfrak{P}\ \forall q \in \mathfrak{R}_{\leq \min(r,r')} & (4.16)\\
& 1 - x^p_{(r,s)} & \geq \sum_{p'\in\mathfrak{P}_{>p}} x^{p'}_{(r,s')} & \quad \forall p \in \mathfrak{P} \forall r \in \mathfrak{R}\ \forall s \in [k]\ \forall s' \in [k]_{<s} & (4.17)\\
& x^p_{(r,s)} & \in \{0,1\} & \quad \forall p \in \mathfrak{P}\ \forall (r,s) \in \mathfrak{S} & (4.18)\\
& t^{\text{arr}}_{q,(r,s)} & \in \mathbb{Q}_{\geq 0} & \quad \forall (r,s) \in \mathfrak{S}\ \forall q \in \mathfrak{R}_{\leq r} & (4.19)\\
& t^{\text{leave}}_{q,(r,s)} & \in \mathbb{Q}_{\geq 0} & \quad \forall (r,s) \in \mathfrak{S}\ \forall q \in \mathfrak{R}_{\leq r} & (4.20)\\
& C_{\max} & \in \mathbb{Q}_{\geq 0} & & (4.21)
\end{aligned}
$$

The optimal seat assignment σ is encoded using binary indicator variables $x^p_{(r,s)}$ for all passengers $p \in \mathfrak{P}$ and seats $(r,s) \in \mathfrak{S}$, as defined in (4.18). This is different from the previous formulation, where passengers were merely assigned to rows, since in this formulation we need a way to tell the seats within a row apart as they represent different entities passing through the plane.

Arrival and leave times for each row $q \in \mathfrak{R}$ and seat $(r, s) \in \mathfrak{S}$ are defined in (4.19) and (4.20) as continuous, although they are guaranteed to be integral, in order to not constraint the solver. The same is true for the makespan variable defined in (4.21). Arrival and leave times are coupled in (4.13).

To ensure a valid assignment, we require that every passenger is assigned exactly one seat in (4.11) and that no seat is assigned to two passengers in (4.12). The makespan condition only needs to take the leave times of each seats predetermined row into account, hence resulting in the formulation in (4.10).

The minimum walking and stowing times defined for each passenger in the instance must be transferred to the seats they are assigned to. This happens in (4.14) for walking and in (4.15) for stowing. Note that only the constraint for rows that must be walked through is an inequality in order to allow staying in a row should the next one still be occupied. The stowing constraint can be an equality since the seat is no longer an obstruction once it has arrived in its position.

Since the passengers from the instance have a defined order and the virtual seats we send through the plane do not, we use constraint (4.16) to transfer the passenger order to the seats and enforce exclusive use of each row at any given time by requiring all seats assigned to previous passengers to have left a row before entering it. As there potentially is a very large number of these constraints, they are added lazily in the actual implementation.

To improve computational performance, we require passengers to occupy rows from one side of the plane to the other according to passenger order in (4.17). This removes the issue caused by the fact that every permutation of passengers within a row results in the same makespan and hence for every solution there are a number of solutions of identical quality, which is disadvantageous for the branch and bound process used to solve the MIP.

Chapter 5

Heuristics

Since the seat assignment problem in aeroplane boarding is NP-hard, as shown in chapter 3, we can expect difficulties computing optimal solutions for reasonably-sized instances. It is hence interesting to look at heuristic methods of producing high-quality solutions quickly. Short computation times are especially important when imagining real-world applications where seat assignment can only happen once passengers have formed a queue and wait to be seated. This chapter will focus on describing existing heuristics from the literature on similar problems, as well as our own heuristics. The results of a computational study comparing computation times and objective values of different heuristics and exact methods will be presented in chapter 6.

Heuristic 1 (Back to front)**.** Boarding passengers back-to-front is an intuitive heuristic. It involves sending each passenger to the unassigned seat that is the furthest to the back of the plane. Since our model does not take passenger interference within rows and outside of the aisle into account, the order in which seats within a row are assigned to passengers is irrelevant. We refer to this heuristic by $\mathcal{H}_{\text{btf}}$.

Heuristic 2 (Window-to-Window)**.** Given a plane layout with k seats, we split passengers into k groups. The first group gets assigned the first seat in each row back-to-front. This process is repeated k times in total. We refer to this heuristic by $\mathcal{H}_{\text{wtw}}$.

Theorem 2. Consider a *BAP* instance as defined in 1. If the plane is full and there exist $m, s \in \mathbb{Q}_{\geq 0}$ such that $t^w_{p,r} = m$ and $t^s_{p,r} = s$ for all passengers $p \in \mathfrak{P}$ and rows $r \in \mathfrak{R}$, then $\mathcal{H}_{\text{wtw}}$ is an optimal boarding strategy.

Proof. We compute the makespan of $\mathcal{H}_{\text{wtw}}$ on such an instance. The boarding process can be split into k phases, each consisting of $|\mathfrak{R}|$ passengers walking to

49	51	53	·	54	52	50
43	45	47		48	46	44
37	39	41		42	40	38
31	33	35		36	34	32
25	27	29		30	28	26
19	21	23		24	22	20
13	15	17		18	16	14
7	9	11		12	10	8
1	3	5		6	4	2

Figure 5.1: A schematic representation of $\mathcal{H}_{\text{btf}}$. The dot $(\cdot)$ indicates the front.

9	27	45	·	54	36	18
8	26	44		53	35	17
7	25	43		52	34	16
6	24	42		51	33	15
5	23	41		50	32	14
4	22	40		49	31	13
3	21	39		48	30	12
2	20	38		47	29	11
1	19	37		46	28	10

Figure 5.2: A schematic representation of $\mathcal{H}_{\text{wtw}}$. The dot $(\cdot)$ indicates the front.

their seats and stowing. Since all passengers have the same walking speeds, they never obstruct each other while walking and since seats within a group are assigned back-to-front no passenger can be obstructed by a stowing passenger from their own boarding group. Since there are $|\mathfrak{R}| - 1$ rows that need to be walked through by the first passenger of each boarding group and all passengers from one group finish boarding at the same time we get the following total boarding time:

$$k((|\mathfrak{R}| - 1) \cdot m + s) \tag{5.1}$$

To prove that $\mathcal{H}_{\text{wtw}}$ is optimal in the scenario at hand, we show that its makespan is equal to a lower bound to the makespan of any boarding strategy. As boarding cannot be finished while there is still a passenger in the aisle by the first row, we find a lower bound on how long this aisle space has to be occupied in any seat assignment. Since k passengers have to be assigned seats in the first row and another $k \cdot (|\mathfrak{R}| - 1)$ have to past it to their seats further back in the plane, the aisle space is occupied for at least the following amount of time:

$$k \cdot s + k \cdot (|\mathfrak{R}| - 1) \cdot m \tag{5.2}$$

□

Heuristic 3 (Reverse Pyramid)**.** The idea of boarding passengers in a *reverse pyramid* scheme was developed in [Bri+05] in an attempt to speed up the boarding process for America West Airlines. It is the result of using mixed integer programming to minimise the number of interferences between passengers during boarding on a series of test instances and manually inferring a general pattern in the results. The resulting pattern is described as a reverse pyramid due to its visual appearance and in [Bri+05] is only documented in terms of a boarding sequence graph for a specific plane layout.

Our implementation of the scheme is inferred from this graph and is applicable to different plane layouts and passenger numbers. Like the original author, we split passengers into six equal-sized boarding groups. The first group is boarded back-to-front in the outermost columns of the seat layout. All other boarding groups are split into two groups in a four to six ratio with the first group again being boarded in the outermost available column back-to-front, and the second group being boarded one column closer to the aisle. Should there not be an available seat in the desired column at any point, the passenger is shifted one column towards the aisle. We refer to this heuristic by $\mathcal{H}_{\text{rev}}$.

16	43	48	·	54	49	23
15	31	47		53	38	22
7	30	46		52	37	14
6	29	45		51	36	13
5	21	44		50	28	12
4	20	35		42	27	11
3	19	34		41	26	10
2	18	33		40	25	9
1	17	32		39	24	8

Figure 5.3: A schematic representation of $\mathcal{H}_{\text{rev}}$. The dot (·) indicates the front.

Heuristic 4 (Steffens method)**.** This method was presented in [Ste08] and results from running a Markov Chain Monte Carlo optimisation algorithm on a set of instances and interpreting the results to manually extract a pattern. The resulting suggested boarding strategy was presented in the form of a figure representing a boarding sequence for a given instance, as was the case for heuristic 3.

Our implementation is inferred from the graphical representation in the paper to be applicable to different size planes. We first assign the leftmost seat in the last row and continue assigning the leftmost seat in every other row back-to-front. The same process is repeated with the rightmost column in the seat layout. Following this, the gaps left in the leftmost column during the first pass are filled back-to-front. Again, the same process is repeated for the rightmost column. Once the outermost columns are filled, we apply the entire procedure to the two columns one seat closer to the aisle. This entire process is repeated iteratively until all seats are filled. We refer to this heuristic by $\mathcal{H}_{\text{Steff}}$.

5	23	41	·	46	28	10
14	32	50		54	36	18
4	22	40		45	27	9
13	31	49		53	35	17
3	21	39		44	26	8
12	30	48		52	34	16
2	20	38		43	25	7
11	29	47		51	33	15
1	19	37		42	24	6

Figure 5.4: A schematic representation of $\mathcal{H}_{\text{Steff}}$. The dot $(\cdot)$ indicates the front.

Heuristic 5 (Local 2-opt search)**.** Given an existing solution to the seat assignment problem, we can attempt to improve it by locally transforming it into a *locally 2-optimal solution*. We call a solution *2-optimal* if its makespan cannot be improved by swapping two passengers' seat assignments. We also refer to this heuristic by $\mathcal{H}_{\text{loc}}$.

In our implementation, we check for every passenger whether swapping seats with any preceding passenger would result in a better makespan. If so, the swap is immediately committed without starting the search from the beginning. The purpose of this strategy is to make as many swaps as possible in any given sweep of the passengers with the goal of improving solutions quickly. The search finishes once no more makespan-improving seat assignment swaps can be found.

In order to perform a local 2-opt search, one needs to repeatedly compute the makespan of a given solution. Since the repeated makespan computation is performance critical, rather than formulating it as a mixed integer program, it is computed directly using the following algorithm:

```python
    def simulate_seating(self) -> SeatingSimulation:
        """
        Computes a seating simulation for this aeroplane boarding solution.
        This includes computation of the makespan.
        """
        bap = self.problem
        passenger_seated_times = [0 for _ in range(bap.num_passengers)]
        passenger_enters_row = []
        row_blockage = [0 for _ in range(bap.num_rows)]

        for passenger in range(bap.num_passengers):
            assigned_row = self.assignment[passenger]
            passenger_enters_row.append([0 for _ in range(assigned_row + 1)])

            for row in range(assigned_row + 1):
                passenger_enters_row[passenger][row] = (
                    row_blockage[0]
                    if row == 0
                    else max(
                        passenger_enters_row[passenger][row - 1]
                        + bap.walking_speeds[passenger][row - 1],
                        row_blockage[row],
                    )
                )

                if not row == 0:
                    row_blockage[row - 1] = passenger_enters_row[passenger][row]

                if row == assigned_row:
                    passenger_seated_time = (
                        passenger_enters_row[passenger][row]
                        + bap.stowing_speeds[passenger][row]
                    )

                    row_blockage[row] = passenger_seated_time
                    passenger_seated_times[passenger] = passenger_seated_time

        makespan = max(passenger_seated_times, default=0)
        seating_simulation = SeatingSimulation(
            passenger_seated_times, passenger_enters_row, makespan, solution=self
        )
        return seating_simulation
```

Chapter 6

Computational Study

In order to conduct a computational study on the performance of the MIP formulations discussed in section 4, we need a set of sample instances. As this thesis discusses a variation of the boarding problem presented in [WT19], we deemed it fitting to adapt the dataset used in that paper for our purposes and generate an additional set using a similar method. This allows us to compare the computational performance of the two problems on the same instances.

6.0.1 Instance Generation

Instances are generated to agree with the choices made in [WT19]. That means that walking times are independently sampled from $\{1, 2, 3\}$ with respective probabilities $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$ and stowing times are obtained by sampling a $z \sim \mathcal{N}(60, 20)$ from a Gaussian distribution and computing the stowing time as $\max\{\{\min\lfloor z\rceil, 120\}, 1\}$.

The given instances consider the four different plane configurations $(10, 2)$, $(20, 2)$, $(20, 4)$ and $(30, 6)$, where the components correspond to the number of rows and the number of seats per row respectively. While passenger times do not vary per row in these instances sets, they vary per passenger - the set `m_p_s` has individual walking times, `m_s_p` has individual stowing times and `m_p_s_p` has both.

Since our problem requires walking and stowing times for each combination of passengers and rows, while the problem instances from [WT19] only require these times up to the assigned row, we enhanced the instances by repeating the last provided value for any given passenger where possible and choosing new ones using the procedure previously described when needed. As the instances from [WT19] do not vary walking or loading times between rows, enhancing the instances in this manner maintains their structure.

Since our problem formulation allows for different walking and stowing times for every combination of passengers and rows, and the given instances do not make use of this feature, we generated another set of instances called `own` of ten instances for each of the four configurations with variable passenger times per row.

6.0.2 Test Description

For each instance, we run three computations - a local search test, the heuristic methods as defined in chapter 5 and four different mixed integer programs. Every time a method has generated a solution, we run a local improvement on it and perform a stability test. The best heuristically computed seat assignment is used as a start solution for those MIPs that have a warm start.

The first computational test consists of generating 100 random solutions and using each of these solutions as a start point for a local search as defined in 5 to find a 2-optimal solution. The intention is to compare the quality of these easily obtained solutions to the optimum or the best bound and to investigate whether generating this many start solutions has any substantial advantage over generating just a few. We refer to this strategy by $\mathcal{H}_{\text{loc}}^{100}$.

The second set of computations comprises the four heuristic strategies *Back-to-Front* (1), *Window-to-Window* (2), *Reverse Pyramid* (3) and STEFFEN*'s Method* (4). The intention here is to compare the solution quality of these heuristics, as they are computationally inexpensive and can be used in real world applications with little effort. We denote these strategies by $\mathcal{H}_{\text{btf}}$, $\mathcal{H}_{\text{wtw}}$, $\mathcal{H}_{\text{rev}}$ and $\mathcal{H}_{\text{Steff}}$ respectively, as defined in chapter 5.

The final set of computations consists of running four MIPs in an attempt to find an exact solution for each instance. The first three MIPs are variations of the *standard formulation* presented in 4.1. The first MIP test simply uses the standard MIP with the best heuristic solution for warm starting. We refer to this strategy by $\mathcal{M}_{\text{std}}$. The second MIP test is similar but uses the additional cuts defined in constraint 4.6. The intention here is to investigate whether these additional cuts help the solver with finding better dual bounds or whether on the contrary, the larger number of constraints in the model leads to inferior solution behaviour. We refer to this strategy by $\mathcal{M}_{\text{std}}^{\text{cuts}}$. In the third MIP test we attempt to solve the instances using the standard MIP from 4.1 without any precomputed initial solutions. We refer to this test by $\mathcal{M}_{\text{std}}^{\text{cold}}$. The intention is to compare the performance of warm-started and cold-started MIPs to investigate whether providing an initial solution aids the branch and bound process by allowing the solver to prune branches early or whether on the contrary our way of generating locally 2-optimal solutions produces seat assignments that hinder the exact solution process by having a structure that leads the solver to search the branch and bound tree in a disadvantageous manner. The final MIP is a warm started version of the *alternative formulation* as presented in 4.2. We refer to this strategy by $\mathcal{M}_{\text{alt}}$. The intention of these computations is to compare the performance of the individual MIPs with respect to the achieved solution quality, the duality gap and computation time.

Regardless the method, every solution was subjected to the local improvement procedure and stability tests once it was generated. This gives us the opportunity to compare how much room for easy improvement the different solution methods leave. The stability tests were comprised of four individual tests, each performed for every passenger in the instance. The first test $\mathcal{S}_{\text{del}}$ delays the selected passenger, that is pushes them to the back of the queue, the second $\mathcal{S}_{\text{swap}}$ swaps the queue position of the selected passenger and a randomly selected passenger, the third $\mathcal{S}_{\text{cng}}$ changes the walking and stowing speeds of a passenger by choosing them according to the procedure described in 6.0.1 and the final test $\mathcal{S}_{\text{all}}$ does all of these things. We call these tests *disturbance strategies.*

6.0.3 Software and Hardware used

Instance generation, parsing, formatted output as well as heuristics were implemented in *Python 3.6* and run in *CPython.* All mixed integer program solving was done using the commercial solver *Gurobi 8.1.0*, using the provided *Python* interface. The experiments were run on 64 cluster nodes, all of which were *HP ProLiant SE316M1* machines equipped with an *Intel Xeon L5630 Quad Core 2.13 GHz* processor. All but eight of these machines were outfitted with 16 GB of RAM and were used for all but the large $(60, 3)$ instances, which ran on machines with 128 GB of RAM. In order to get results within a reasonable amount of time and to investigate possible real world usage, we configured the MIP solver to time out after two hours.

6.1 Results

The tables 6.1 through 6.4 show the results of our computational experiments for each of the sets of instances described in section 6.0.1. Each of these sets consists of 40 instances, ten for each of the four seat layout configurations. The values presented here are the average values over all 40 instances in a set.

For each strategy (with the exception of the alternative MIP), we present the average makespan of the solutions generated by the strategy as well as the average makespan of these solutions after being used as a starting point for the local improvement strategy $\mathcal{H}_{\text{loc}}$ in the column labeled *2-opt.* The average improvement between these two seat assignments is expressed as a percentage in the column labeled *% imp.*

For the MIP-based strategies, we present the average best gap and the number of instances that could be solved optimally within a time frame of two hours. Since the heuristic strategies cannot be used to prove optimality, these data points are not provided there. Similarly, the solve time for the heuristic methods is so short that it is dominated by factors like loading the instances from disk and starting the Python runtime.

The alternative MIP as defined in section 4.2 turned out to be very resource-intensive in the solving process. For all but the small $(10, 2)$ seat layout configurations, the solver crashed due to insufficient RAM. Hence, the average values for the makespan cannot be compared to the ones for the other strategies and no local optimisation results are given since the local improvement did not run after running out of memory.
Since the alternative MIP formulation proved to be infeasible due to memory constraints, we will disregard it for most of the discussion. The most striking result is how well the heuristic $\mathcal{H}_{\text{wtw}}$ performs - not only is it the best of all heuristic approaches and appears to produce locally optimal solutions, but the exact methods can barely improve over these results. Among the three variations of the standard MIP formulation, $\mathcal{M}_{\text{std}}^{\text{cold}}$ is significantly outperformed by the other variety. Between the warm-started plain variety $\mathcal{M}_{\text{std}}$ and the warm-started version with extra cuts $\mathcal{M}_{\text{std}}^{\text{cuts}}$, there is a small difference depending on the instance set - for `m_p_s` instances, $\mathcal{M}_{\text{std}}$ is slightly better with regard to all performance parameters, while for the other instance sets, $\mathcal{M}_{\text{std}}^{\text{cuts}}$ trades a longer average solve time for slightly better average makespans and gaps.

6.1.1 Dual Gaps for MIPs

The figures 6.1 and 6.2 show the development of the best integer solution and the best dual bounds for the three varieties of the standard MIP over time. Since the alternative MIP formulation ran out of memory on most of the instances and timed out on all, it was omitted from these visualisations.

The figures show values for the strategies $\mathcal{M}_{\text{std}}$ in red, $\mathcal{M}_{\text{std}}^{\text{cuts}}$ in blue and $\mathcal{M}_{\text{std}}^{\text{cold}}$ in green, with integer incumbents shown as squares and dual bounds as diamonds. The solve time in seconds is on the x-axis and objective values in seconds are on the y-axis.

The solve behaviour for the instance `own_10_2_0.abp` shown in figure 6.1 is representative for most instances that have an optimal solution that is different from the $\mathcal{H}_{\text{wtw}}$ solution. Here, the warm started MIPs seem to be at a disadvantage compared to $\mathcal{M}_{\text{std}}^{\text{cold}}$, especially at finding good solutions and even manages to raise the dual bound slightly faster than $\mathcal{M}_{\text{std}}$ and $\mathcal{M}_{\text{std}}^{\text{cuts}}$, which seems to be slower on these kinds of instances.

Since the $\mathcal{H}_{\text{wtw}}$ solution used for warm starting was already optimal for the instance `m_p_s_p_10_2_0.abp`, there is no development of the best integer solution in figure 6.2 for $\mathcal{M}_{\text{std}}$ and $\mathcal{M}_{\text{std}}^{\text{cuts}}$. On this instance, the vanilla standard formulation $\mathcal{M}_{\text{std}}$ manages to close the duality gap the fastest, followed by $\mathcal{M}_{\text{std}}^{\text{cuts}}$ and $\mathcal{M}_{\text{std}}^{\text{cold}}$.

Table 6.1: Computation results for `m_p_s` instances (mean over 40 instances of all configurations)

strategy	makespan (s)	2-opt (s)	% imp	% gap	# opt	time (s)
$\mathcal{H}_{\mathrm{loc}}^{100}$	662.0	662.0	0.0	-	-	-
$\mathcal{H}_{\mathrm{btf}}$	3523.3	760.78	70.53	-	-	-
$\mathcal{H}_{\mathrm{wtw}}$	423.53	423.53	0.0	-	-	-
$\mathcal{H}_{\mathrm{rev}}$	2170.88	750.23	63.78	-	-	-
$\mathcal{H}_{\mathrm{Steff}}$	632.35	621.0	2.11	-	-	-
$\mathcal{M}_{\mathrm{std}}$	423.53	423.53	0.0	6.82	20	4565.98
$\mathcal{M}_{\mathrm{std}}^{\mathrm{cuts}}$	423.53	423.53	0.0	7.31	19	4694.7
$\mathcal{M}_{\mathrm{std}}^{\mathrm{cold}}$	1110.45	672.28	16.66	38.92	10	5419.35
$\mathcal{M}_{\mathrm{alt}}^{*}$	208.85	-	-	41.50	0	7200.0

Table 6.2: Computation results for `m_s_p` instances (mean over 40 instances of all configurations)

strategy	makespan (s)	2-opt (s)	% imp	% gap	# opt	time (s)
$\mathcal{H}_{\mathrm{loc}}^{100}$	659.08	659.08	0.0	-	-	-
$\mathcal{H}_{\mathrm{btf}}$	3694.35	750.5	71.64	-	-	-
$\mathcal{H}_{\mathrm{wtw}}$	481.73	481.5	0.043	-	-	-
$\mathcal{H}_{\mathrm{rev}}$	2279.8	757.6	64.16	-	-	-
$\mathcal{H}_{\mathrm{Steff}}$	752.25	711.1	6.33	-	-	-
$\mathcal{M}_{\mathrm{std}}$	481.0	481.0	0.0	25.87	13	5193.36
$\mathcal{M}_{\mathrm{std}}^{\mathrm{cuts}}$	480.13	480.13	0.0	25.70	12	5346.08
$\mathcal{M}_{\mathrm{std}}^{\mathrm{cold}}$	1179.08	702.25	19.32	49.02	10	5414.7
$\mathcal{M}_{\mathrm{alt}}^{*}$	235.8	-	-	52.51	0	7200.0

Table 6.3: Computation results for `m_p_s_p` instances (mean over 40 instances of all configurations)

strategy	makespan (s)	2-opt (s)	% imp	% gap	# opt	time (s)
$\mathcal{H}_{\text{loc}}^{100}$	681.98	681.98	0.0	-	-	-
$\mathcal{H}_{\text{btf}}$	3693.15	768.65	71.17	-	-	-
$\mathcal{H}_{\text{wtw}}$	516.58	514.7	0.20	-	-	-
$\mathcal{H}_{\text{rev}}$	2261.55	756.73	63.92	-	-	-
$\mathcal{H}_{\text{Steff}}$	778.85	726.83	7.64	-	-	-
$\mathcal{M}_{\text{std}}$	512.98	512.98	0.0	25.79	12	5212.83
$\mathcal{M}_{\text{std}}^{\text{cuts}}$	512.93	512.93	0.0	25.54	12	5237.45
$\mathcal{M}_{\text{std}}^{\text{cold}}$	1240.95	740.18	20.13	49.35	10	5424.23
$\mathcal{M}_{\text{alt}}^{*}$	251.7	-	-	56.93	0	7200.0

Table 6.4: Computation results for `own` instances (mean over 40 instances of all configurations)

strategy	makespan (s)	2-opt (s)	% imp	% gap	# opt	time (s)
$\mathcal{H}_{\text{loc}}^{100}$	546.55	546.55	0.0	-	-	-
$\mathcal{H}_{\text{btf}}$	3693.83	623.23	75.46	-	-	-
$\mathcal{H}_{\text{wtw}}$	496.23	485.38	1.97	-	-	-
$\mathcal{H}_{\text{rev}}$	2247.4	617.43	69.75	-	-	-
$\mathcal{H}_{\text{Steff}}$	766.7	597.48	22.20	-	-	-
$\mathcal{M}_{\text{std}}$	480.55	480.33	0.08	27.92	10	5420.75
$\mathcal{M}_{\text{std}}^{\text{cuts}}$	480.15	479.98	0.06	27.77	10	5424.8
$\mathcal{M}_{\text{std}}^{\text{cold}}$	857.53	561.15	17.79	43.88	10	5434.05
$\mathcal{M}_{\text{alt}}^{*}$	238.1	-	-	64.94	0	7200.0

Figure 6.1: Best integer solutions (squares) and best dual bounds (diamonds) for `own_10_2_0.abp` using $\mathcal{M}_{\text{std}}$ (red), $\mathcal{M}_{\text{std}}^{\text{cuts}}$ (blue) and $\mathcal{M}_{\text{std}}^{\text{cold}}$ (green). Time in seconds on x-axis, objective value in seconds on y-axis.

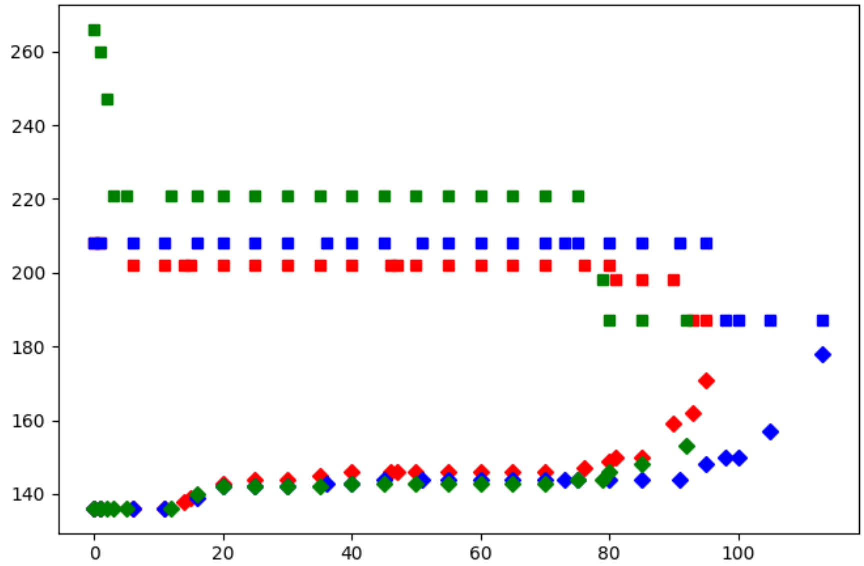
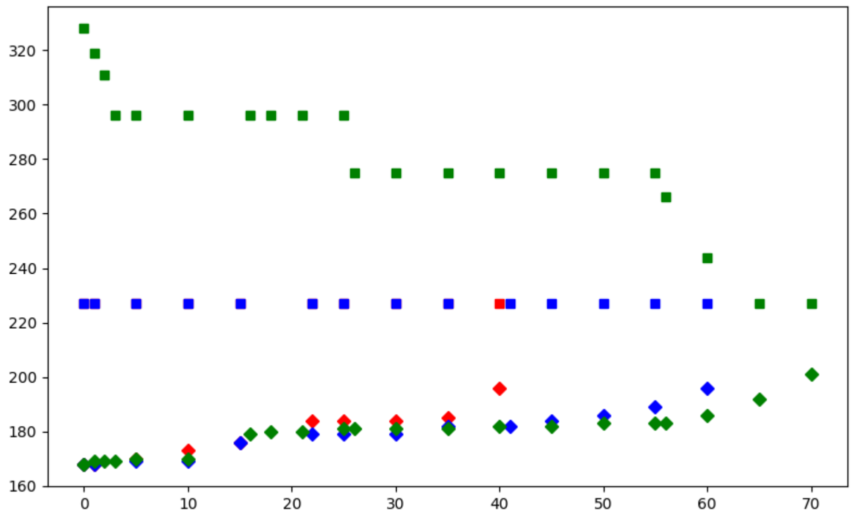


Figure 6.2: Best integer solutions (squares) and best dual bounds (diamonds) for `m_p_s_p_10_2_0.abp` using $\mathcal{M}_{\text{std}}$ (red), $\mathcal{M}_{\text{std}}^{\text{cuts}}$ (blue) and $\mathcal{M}_{\text{std}}^{\text{cold}}$ (green). Time in seconds on x-axis, objective value in seconds on y-axis.



6.1.2 Robustness of Solutions

Tables 6.5 and 6.6 show the results of the stability tests as defined in section 6.0.2 on the sets `own` and `m_p_s_p`, respectively. The starting point for every strategy is a 2-opt solution generated using the given strategy and $\mathcal{H}_{\text{loc}}$ and all values are means over 40 instances from the given instance set. Every one of the disturbance strategies $\mathcal{S}_{\text{del}}$, $\mathcal{S}_{\text{swap}}$, $\mathcal{S}_{\text{cng}}$ and $\mathcal{S}_{\text{all}}$ is run once with every passenger as the affected of the disturbance and the individual resulting makespans are averaged.

The results for both instance sets are consistent in that disturbing the seat assignments generated by different strategies maintains their order in terms of quality, that is better start solutions remain better after disturbance. Regardless of the strategy that generated the start solution, the amount of negative impact the disturbance strategies have is least for $\mathcal{S}_{\text{cng}}$, followed by $\mathcal{S}_{\text{del}}$ and $\mathcal{S}_{\text{swap}}$ with $\mathcal{S}_{\text{all}}$ having the biggest negative impact.

6.1.3 A non-optimal $\mathcal{H}_{\text{wtw}}$ Solution

As was shown in theorem 2, the $\mathcal{H}_{\text{wtw}}$ heuristic generates optimal seat assignments when all passengers have equal walking and stowing times. Since stowing times are much bigger than walking times in our instances, the instance set `m_p_s` where stowing times are the same for all passengers exhibits the same behaviour. For those instances where stowing times are individual to each passenger, it still appears to be an excellent heuristic to pretend they are all the same.

There are however a handful of instances in our test sets for which a $\mathcal{H}_{\text{wtw}}$ solution is not optimal. Figures 6.3 and 6.4 respectively visualise the boarding process for a $\mathcal{H}_{\text{wtw}}$ and $\mathcal{M}_{\text{std}}$ solution for the instance `own_10_2_0.abp`. In these visualisations, the x-axis corresponds to the rows in the plane and the y-axis corresponds to time in seconds. Each coloured line represents a passenger making their way through the plane. As one can see, the reduction in makespan by the optimal solution compared to the heuristic one was not achieved by merely swapping two passengers, but by a more complex reordering.

6.1.4 Comparison to Boarding Sequence Optimisation

Since [WT19] investigated a related problem where they got to chose boarding order rather than seat assignment and tested their implementations on the same set of instances, it is worth comparing the results of their computational study to ours.

The makespans for the solutions generated by exact MIP methods were generally slightly better in the scenario from [WT19] - for `m_p_s` they got 395.6 seconds on average versus our 423.53 seconds, for `m_s_p` it was 429.1 seconds versus 480.13

Table 6.5: Stability tests as defined in section 6.0.2 on `own` instances (mean over 40 instances of all configurations)

strategy	makespan (s)	$\mathcal{S}_{\text{del}}$	$\mathcal{S}_{\text{swap}}$	$\mathcal{S}_{\text{cng}}$	$\mathcal{S}_{\text{all}}$
$\mathcal{H}_{\text{loc}}^{100}$	546.55	574.29	593.45	561.28	630.54
$\mathcal{H}_{\text{btf}} + \mathcal{H}_{\text{loc}}$	623.23	656.32	670.47	635.24	705.88
$\mathcal{H}_{\text{wtw}} + \mathcal{H}_{\text{loc}}$	485.38	533.6	567.83	486.71	609.06
$\mathcal{H}_{\text{rev}} + \mathcal{H}_{\text{loc}}$	617.43	650.23	663.06	629.37	699.87
$\mathcal{H}_{\text{Steff}} + \mathcal{H}_{\text{loc}}$	597.48	632.28	652.56	606.25	686.82
$\mathcal{M}_{\text{std}} + \mathcal{H}_{\text{loc}}$	480.33	524.07	551.26	485.18	597.38

Table 6.6: Stability tests as defined in section 6.0.2 on `m_p_s_p` instances (mean over 40 instances of all configurations)

strategy	makespan (s)	$\mathcal{S}_{\text{del}}$	$\mathcal{S}_{\text{swap}}$	$\mathcal{S}_{\text{cng}}$	$\mathcal{S}_{\text{all}}$
$\mathcal{H}_{\text{loc}}^{100}$	681.98	725.52	756.28	687.05	795.56
$\mathcal{H}_{\text{btf}} + \mathcal{H}_{\text{loc}}$	768.65	812.58	831.32	774.3	872.27
$\mathcal{H}_{\text{wtw}} + \mathcal{H}_{\text{loc}}$	514.7	563.88	594.28	514.85	637.98
$\mathcal{H}_{\text{rev}} + \mathcal{H}_{\text{loc}}$	756.73	801.32	822.48	761.51	863.91
$\mathcal{H}_{\text{Steff}} + \mathcal{H}_{\text{loc}}$	726.83	768.21	790.2	729.3	823.66
$\mathcal{M}_{\text{std}} + \mathcal{H}_{\text{loc}}$	512.96	561.47	593.13	514.34	633.94

Figure 6.3: An $\mathcal{H}_{\mathrm{wtw}}$ solution for `own_10_2_0.abp`. Plane rows on x-axis, time in seconds on y-axis.

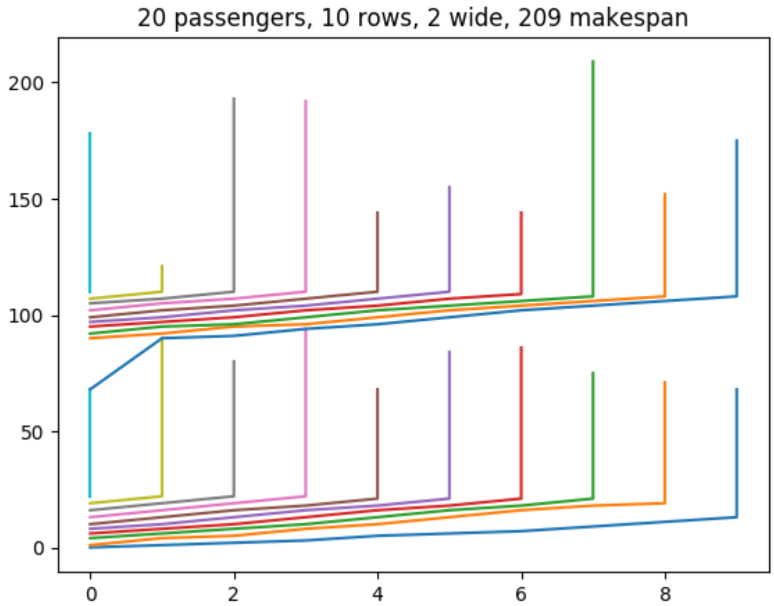
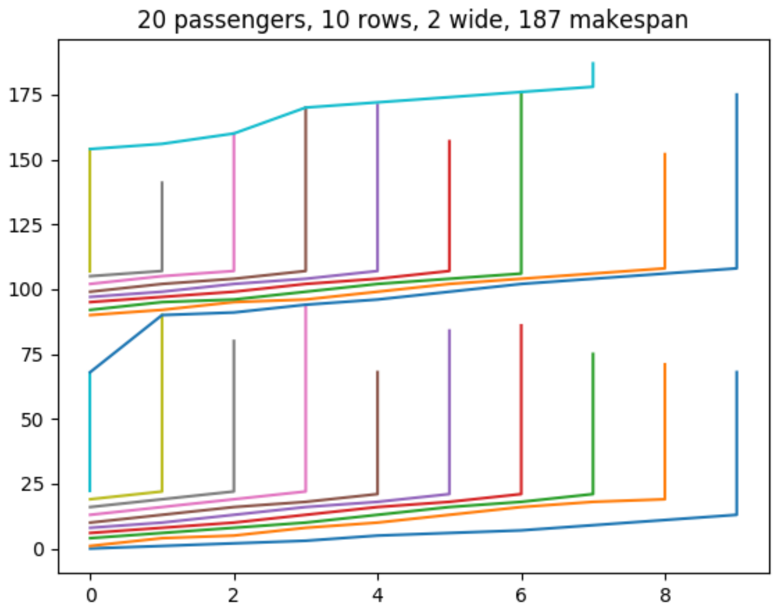


Figure 6.4: A $\mathcal{M}_{\mathrm{std}}$ solution for `own_10_2_0.abp`. Plane rows on x-axis, time in seconds on y-axis.



seconds and for `m_p_s_p` it was 452.1 seconds versus 512.93 seconds. Not only did out MIP approaches produce higher makespans, they were also more expensive computationally, taking longer to solve on average and finishing fewer instances optimally within the two hour time limit. However it should be mentioned that the benchmark instances were constructed in such a way that the stowing time was identical for all seats for any given passenger, reducing the potential for optimisation in out approach.

The behaviour the solutions exhibit when disturbed are similar for both approaches, which should be expected, since the makespan is defined in the same way. Another common finding for both approaches is that the work invested in finding optimal solutions using mixed integer programming only results in solutions that are barely superior to those found using a simple heuristic, which for us is $\mathcal{H}_{\text{wtw}}$ and for [WT19] is their *max-settle-row* strategy.

6.2 Conclusions

The first observation is that the alternative MIP formulation from section 4.2 is not functional in its current form and might profit from improved heuristics to find better constraints to lazily add to the model when a new MIP incumbent is found.

More importantly, the difference in terms of makespan between the solutions produced by the best performing exact strategy $\mathcal{M}_{\text{std}}^{\text{cuts}}$ and the best performing heuristic $\mathcal{H}_{\text{wtw}}$ is miniscule on almost all instances that were tested. What that means in practice is that investing multiple hours of computation time into optimisation using mixed integer programs only results in boarding times that are a few seconds shorter than those produced by $\mathcal{H}_{\text{wtw}}$ almost instantly. The difference between exact and heuristic solutions is generally much smaller than the potential changes in makespan that might result from disturbances such as a delayed passenger, which further diminishes the value of using MIP approaches at this point.

Chapter 7

Outlook and Future Research

This chapter lays out a few pointers and research opportunities that have become apparent during the creation of this thesis but were unfortunately out of scope for various reasons.

7.1 Heuristics and Data

As was shown in theorem 2, $\mathcal{H}_{\text{wtw}}$ is an optimal seat assignment strategy if all passengers have the same walking and stowing times for all rows. Since the results of the computational study in section 6.1 indicate that the heuristic is actually optimal for a broader set of instances, it might be of interest to establish a more solid theory of the characterisation of instances for which $\mathcal{H}_{\text{wtw}}$ is optimal.

Whether $\mathcal{H}_{\text{wtw}}$ is an optimal strategy most likely depends on the distribution of individual passenger times, especially how similar they are to one another and how big the difference between walking and stowing times is. It might therefore be of special interest to find a reliable source of representative real world data for these times.

One aspect of such data that has the potential of determining whether $\mathcal{H}_{\text{Steff}}$ or $\mathcal{H}_{\text{wtw}}$ is superior is the behaviour of passengers when standing in neighbouring aisle spaces and stowing. In our model, we assume that such a situation does not slow down the stowing process while [Ste08] assumed the opposite.

7.2 Online Setting

In real-life scenarios, passengers will probably not form a perfect queue before entering the plane. Rather, they might crowd in front of the entrance and enter one by one, which in mathematical terms, we can consider a queue that we only learn one passenger at a time. This observation naturally motivates a more thorough study of online variants of *BAP*, which was outside the scope of this thesis. What we present here are a few online scenarios that are interesting and yet easy to analyse.

7.2.1 Possible Scenarios

We consider a modified version *online BAP-1* of the problem, in which the online procedure does not know the number of passengers and only has access to the movement and stowing speed of the frontmost passenger. It is easy to see that for any $j \in \mathbb{N}_{>0}$, there cannot be a j-competitive online algorithm for this modified problem.

Proof. Let $j \in \mathbb{N}_{>0}$. Consider the following family of instances of *online BAP-1*: $\mathfrak{R} = (r_1, r_2), k_1 = 1, k_2 = 0$. The adversary presents the first passenger p_1 with $t^s_{p_1,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_1,1} = 1$. Any online algorithm A for *online BAP-1* falls into one of the following two cases:

1. A assigns $\sigma(p_1) = (r_1, 1)$. The adversary presents a second passenger p_2 with $t^s_{p_2,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_2,1} = j + 1$. Since A can only assign $\sigma(p_2) = (r_2, 1)$, it generates $\text{cost}(A) = j + 1$ while $\text{cost}(\text{opt}) = 1$.

2. A assigns $\sigma(p_1) = (r_2, 1)$. The adversary does not present a second passenger. Hence $\text{cost}(A) = 1$ while $\text{cost}(\text{opt}) = 0$.

$\square$

In the proof above, the adversary used the fact that any online algorithm for *online BAP-1* needs to guess the number of passengers. We now consider another modified problem *online BAP-2*, in which the online algorithm does know the total number of passengers. Again we show that for any $j \in \mathbb{N}_{>0}$ there can be no j-competitive algorithm for *online BAP-2*.

Proof. Let $j \in \mathbb{N}_{>0}$. Consider the following family of instances of *online BAP-2*: $\mathfrak{R} = (r_1, r_2), k_1 = 1, k_2 = 0$ and $|\mathfrak{P}| = 2$. The adversary presents the first passenger p_1 with $t^s_{p_1,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_1,1} = 1$. Any online algorithm A for *online BAP-2* falls into one of the following two cases:

1. A assigns $\sigma(p_1) = (r_1, 1)$. The adversary presents a second passenger p_2 with $t^s_{p_2,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_2,1} = j + 1$. Since A can only assign $\sigma(p_2) = (r_2, 1)$, it generates $\text{cost}(A) = j + 1$ while $\text{cost}(\text{opt}) = 1$.

2. A assigns $\sigma(p_1) = (r_2, 1)$. The adversary presents a second passenger p_2 with $t^s_{p_2,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_2,1} = 0$. Since A can only assign $\sigma(p_2) = (r_2, 1)$, it generates $\text{cost}(A) = 1$ while $\text{cost}(\text{opt}) = 0$.

□

In the proof above, we used the fact that the adversary for *online BAP-2* is not committed to any bounds on the movement speed of the passengers and can present arbitrarily fast or slow passengers. We therefore now consider yet another modified problem *online BAP-3*, in which the online algorithm does know both the total number of passengers and tight bounds $t^s_{L,\mathfrak{R}}, t^s_{U,\mathfrak{R}}$ on stowing speeds and $t^w_{L,\mathfrak{R}}, t^w_{U,\mathfrak{R}}$ on movement speeds. Once again, we show that for any $j \in \mathbb{N}_{>0}$ there can be no j-competitive algorithm for *online BAP-3*.

Proof. Let $j \in \mathbb{N}_{>0}$. Consider the following family of instances for *online BAP-3*: $\mathfrak{R} = (r_1, r_2), k_1 = k_2 = 1, |\mathfrak{P}| = 4, t^s_{L,\mathfrak{R}} = t^s_{U,\mathfrak{R}} = t^w_{L,\mathfrak{R}} = 0$ and $t^w_{U,\mathfrak{R}} = j + 1$. The adversary presents the first passenger p_1 with $t^s_{p_1,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_1,1} = 0$. Any online algorithm A for *online BAP-3* falls into one of the following two cases:

1. A assigns $\sigma(p_1) \in \{(r_1, 1), (r_1, 2)\}$. The adversary continues to present p_2 with $t^s_{p_2,r} = t^w_{p_2,1} = 0$ for all $r \in \mathfrak{R}$ and p_3, p_4 with $t^s_{p_3,r} = t^s_{p_4,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_3,1} = t^w_{p_4,1} = j + 1$. No matter the strategy, A cannot assign p_3 and p_4 to seats in r_1, hence $\text{cost}(A) \geq j + 1$ while $\text{cost}(\text{opt}) = 0$.
2. A assigns $\sigma(p_1) \in \{(r_2, 1), (r_2, 2)\}$. The adversary presents p_2 with $t^s_{p_2,r} = 0$ for all $r \in \mathfrak{R}$ and $t^w_{p_2,1} = j + 1$. Again, any online algorithm A falls into one of the following two cases:
   (a) A assigns $\sigma(p_2) \in \{(r_2, 1), (r_2, 2)\}$. The adversary presents p_3 and p_4 with $t^s_{p_3,r} = t^s_{p_4,r} = t^w_{p_3,1} = t^w_{p_4,1} = 0$ for all $r \in \mathfrak{R}$, yielding $\text{cost}(A) = j+1$ while $\text{cost}(\text{opt}) = 0$.
   (b) A assigns $\sigma(p_2) \in \{(r_1, 1), (r_1, 2)\}$. This leaves one seat in the front row and on in the back. No cost has occurred so far. Continue the proof as for *online BAP-2*.

□

Modifying the problem formulation such that movement times must be strictly positive, replacing 0 with a positive $t^w_{L,\mathfrak{R}}$ yields that the competitive ratio of any online algorithm for the problem cannot be better than $\frac{t^w_{U,\mathfrak{R}}}{3t^w_{L,\mathfrak{R}}}$.

7.2.2 Consequences

Theorem 3. For any instance I of *online-BAP* with finite, non-zero movement and stowing times, i.e. there exist $t_{\mathfrak{L}}, t_{\mathfrak{U}} \in \mathbb{R}_{>0}$ such that $t_{\mathfrak{L}} \leq t \leq t_{\mathfrak{U}}$ for all $t \in \{t^s_{p,r} \mid p \in \mathfrak{P}, r \in \mathfrak{R}\} \uplus \{t^w_{p,r} \mid p \in \mathfrak{P}, r \in \mathfrak{R}\}$ and $|\mathfrak{P}| = |\mathfrak{S}|$, any online seat assignment algorithm A is competitive with the following ratio:

$$\frac{\mathrm{cost}(A)}{\mathrm{cost}(\mathrm{opt})} \leq \frac{t_{\mathfrak{U}}}{t_{\mathfrak{L}}}$$

Proof. Since every seat needs to be utilised, a lower bound on the optimal cost is as follows:

$$\mathrm{cost}(\mathrm{opt}) \geq \frac{kt_{\mathfrak{L}}|\mathfrak{R}|(|\mathfrak{R}|-1)}{2}$$

On the other hand, regardless of the seat assignment, no passenger ever spends longer than $t_{\mathfrak{U}}$ in any given row. If any passenger did so, they would be forced to wait after having already spent $t_{\mathfrak{U}}$ in a row. That would indicate that the passenger in front of them already spent longer than $t_{\mathfrak{U}}$ in that row. Inductively, the first passenger to pass through the row would have to have spent longer than $t_{\mathfrak{U}}$ in that row, which contradicts the definition of $t_{\mathfrak{U}}$. Hence an upper bound on the cost incurred by any seat assignment algorithm A can be given as follows:

$$\mathrm{cost}(A) \leq \frac{kt_{\mathfrak{U}}|\mathfrak{R}|(|\mathfrak{R}|-1)}{2}$$

The remark follows immediately. □

Remark 7.2.1. All heuristics from chapter 5, including the often-cited $\mathcal{H}_{\mathrm{Steff}}$ and the high-performing $\mathcal{H}_{\mathrm{wtw}}$ achieve no better competitive ratio than in theorem 3.

Proof. None of the algorithms from chapter 5 takes the walking and stowing times of the passengers into consideration. Hence they have no more information about the problem instance than any algorithm solving *online BAP-3* and the same proof applies. □

Chapter 8

Conclusion

In this thesis, we have formalised a version of the seat assignment problem in aeroplane boarding in chapter 2. As part of this formulation, we assumed that different passengers can have individually different walking and stowing times, which to the best of our knowledge is unique to this thesis and the work in [WT19].

As is the case with the related problem where one may rearrange passengers, the problem of assigning seats to minimise boarding time is NP-hard, which we have shown in chapter 3. To achieve this, we reduced *3-Partition* to our problem, only utilising differences in stowing times.

Knowing that the problem is hard in theory, we presented various heuristic approaches in chapter 5, including a strategy we call *window-to-window*, which we have shown to be optimal for instances with identical walking and stowing times for all passengers. In addition to heuristic approaches we presented two MIP formulations in chapter 4 that can be used to solve optimally or as approximation schemes.

Our computational study in chapter 6 indicated that the *window-to-window* heuristic produces excellent results that can barely be improved upon using the MIPs in practice. The differences in makespan between the best heuristic solutions and MIP solutions that can be computed within two hours are overshadowed by the increase in makespan that can be caused by a delayed passenger or faulty input data, as indicated by our robustness study in section 6.1.2.

In addition to recommending further research into the characterisation of instances for which *window-to-window* is optimal, in chapter 7 we looked at properties of simple online variations of the seat assignment problem. We showed that any algorithm that does not take the distribution of walking and stowing times into account cannot be competitive in theory, which includes the very well performing *window-to-window* heuristic.

References

[AVB09] Jan Audenaert, Katja Verbeeck and Greet Vanden Berghe. 'Multi-agent based simulation for boarding'. In: *The 21st Belgian-Netherlands Conference on Artificial Intelligence*. 2009, pp. 3–10.

[Bal+10] Michael Ball et al. 'Total delay impact study'. In: *NEXTOR Research Symposium, Washington DC. http://www. nextor. org*. 2010.

[Baz07] Massoud Bazargan. 'A linear programming approach for aircraft boarding strategy'. In: *European Journal of Operational Research* 183.1 (2007), pp. 394–411. ISSN: 0377-2217. DOI: `https://doi.org/10.1016/j.ejor.2006.09.071`. URL: `http://www.sciencedirect.com/science/article/pii/S0377221706010137`.

[Bri+05] Menkes H. L. van den Briel et al. 'America West Airlines Develops Efficient Boarding Strategies'. In: *INFORMS Journal on Applied Analytics* 35.3 (2005), pp. 191–201. DOI: `10.1287/inte.1050.0135`. URL: `https://pubsonline.informs.org/doi/abs/10.1287/inte.1050.0135`.

[CTA04] Andrew J Cook, Graham Tanner and Stephen Anderson. *Evaluating the true cost to airlines of one minute of airborne or ground delay.* Tech. rep. University of Westminster, 2004.

[GJ75] M. Garey and D. Johnson. 'Complexity Results for Multiprocessor Scheduling under Resource Constraints'. In: *SIAM Journal on Computing* 4.4 (1975), pp. 397–411. DOI: `10.1137/0204035`. URL: `https://doi.org/10.1137/0204035`.

[JM17] Shafagh Jafer and Wei Mi. 'Comparative study of aircraft boarding strategies using cellular discrete event simulation'. In: *Aerospace* 4.4 (2017), p. 57.

[JN15] Florian Jaehn and Simone Neumann. 'Airplane boarding'. In: *European Journal of Operational Research* 244.2 (2015), pp. 339–359. ISSN: 0377-2217. DOI: `https://doi.org/10.1016/j.ejor.2014.12.008`. URL: `http://www.sciencedirect.com/science/article/pii/S0377221714009904`.

[Mou11] Jad Mouawad. *Most Annoying Airline Delays Might Just Be in the Boarding*. 2011. URL: https://www.nytimes.com/2011/11/01/business/airlines-are-trying-to-cut-boarding-times-on-planes.html.

[MS16] R. John Milne and Mostafa Salari. 'Optimization of assigning passengers to seats on airplanes based on their carry-on luggage'. In: *Journal of Air Transport Management* 54 (2016), pp. 104–110. ISSN: 0969-6997. DOI: https://doi.org/10.1016/j.jairtraman.2016.03.022. URL: http://www.sciencedirect.com/science/article/pii/S0969699715300235.

[MSK18] R. John Milne, Mostafa Salari and Lina Kattan. 'Robust Optimization of Airplane Passenger Seating Assignments'. In: *Aerospace* 5.3 (2018). ISSN: 2226-4310. DOI: 10.3390/aerospace5030080. URL: http://www.mdpi.com/2226-4310/5/3/80.

[Sch10] Andreas Schlegel. *Bodenabfertigungsprozesse im Luftverkehr: Eine statistische Analyse am Beispiel der Deutschen Lufthansa AG am Flughafen Frankfurt/Main*. 1st ed. Gabler Verlag, 2010. ISBN: 978-3-8349-2399-8,978-3-8349-8691-7. URL: http://gen.lib.rus.ec/book/index.php?md5=894e514572bc804321359b16e9f2b76e.

[SMK19] Mostafa Salari, R. John Milne and Lina Kattan. 'Airplane boarding optimization considering reserved seats and passengers' carry-on bags'. In: *OPSEARCH* 56.3 (Sept. 2019), pp. 806–823. ISSN: 0975-0320. DOI: 10.1007/s12597-019-00405-z. URL: https://doi.org/10.1007/s12597-019-00405-z.

[Ste08] Jason H. Steffen. 'Optimal boarding method for airline passengers'. In: *Journal of Air Transport Management* 14.3 (2008), pp. 146–150. ISSN: 0969-6997. DOI: https://doi.org/10.1016/j.jairtraman.2008.03.003. URL: http://www.sciencedirect.com/science/article/pii/S0969699708000239.

[Sto14] Nick Stockton. *What's up with that: Boarding Airplanes takes forever*. 2014. URL: https://www.wired.com/2014/11/whats-boarding-airplanes-takes-forever/.

[Str14] Joseph Stromberg. *The way we board airplanes makes absolutely no sense*. Apr. 2014. URL: https://www.vox.com/2014/4/25/5647696/the-way-we-board-airplanes-makes-absolutely-no-sense.

[WT19] F.J.L. Willamowski and A.M. Tillmann. *Minimizing Airplane Boarding Time*. repORt 2019–56. Lehrstuhl für Operations Research, RWTH Aachen University, Nov. 2019. URL: https://www.or.rwth-aachen.de/files/research/repORt/2019_Minimizing_Airplane_Boarding_Time_Willamowski_Tillmann.pdf.

Appendix A

Code Listings

A.1 Library

```
import math
import os
import pickle
import random
import sys
from abc import ABC
from contextlib import contextmanager
from copy import deepcopy
from datetime import datetime, timedelta
from itertools import chain, product, repeat, islice, cycle
from pickle import UnpicklingError
from tempfile import NamedTemporaryFile
from typing import Dict, List, Optional, Tuple

import gurobipy
import matplotlib.pyplot as plt
import numpy.random as np_rand
from gurobipy.gurobipy import quicksum
from matplotlib.patches import Rectangle


# utility to suppress gurobi terminal output
@contextmanager
def suppress_stdout():
    with open(os.devnull, "w") as devnull:
        old_stdout = sys.stdout
        sys.stdout = devnull
        try:
            yield
        finally:
```

```
            sys.stdout = old_stdout


GUROBI_LOG_NAME = "bap_mips.log"


class SeatingSimulation:
    """
    Stores simulated seating for a given solution for an aeroplane boarding problem.
    """

    def __init__(
        self,
        passenger_seated_times: List[int],
        passenger_enters_row: List[List[int]],
        makespan: int,
        solution: "BapSolution",
    ):
        self.passenger_seated_times = passenger_seated_times
        self.passenger_enters_row = passenger_enters_row
        self.makespan = makespan
        self.solution = solution

    def __str__(self):
        s = f"makespan: {self.makespan}"
        for passenger in range(len(self.passenger_seated_times)):
            s += "\n"
            s += f"p{passenger} seated at {self.passenger_seated_times[passenger]}: {self.passenger_enters_row[passenger]}"
        return s

    def generate_plot(self):
        num_passengers = len(self.passenger_seated_times)

        fig = plt.figure()
        if self.solution.solver_description is not None and self.solution.computation_time is not None:
            fig.suptitle(
                f"Generated by {self.solution.solver_description} in {self.solution.computation_time}"
            )
        ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
        ax.set_title(
            f"{self.solution.problem.num_passengers} passengers, {self.solution.problem.num_rows} rows, {self.solution.problem.seats_per_row} wide, {self.makespan} makespan"
        )
```

```
        for p in range(num_passengers):
            assigned_row = len(self.passenger_enters_row[p]) - 1
            x = list(range(assigned_row + 1)) + [assigned_row]
            y = self.passenger_enters_row[p].copy() + [self.
    passenger_seated_times[p]]
            ax.plot(x, y)

        return fig


class BapSolution:
    """
    Contains a seat assignment and the makespan.
    """

    def __init__(
        self,
        problem: "AeroplaneBoardingProblem",
        assignment: List[int],
        computation_time: Optional[timedelta] = None,
        solver_description: Optional[str] = None,
        makespan: Optional[float] = None,
        seating_simulation: Optional[SeatingSimulation] = None,
        solver_output: Optional[str] = None,
    ):
        self.assignment = assignment
        self.problem = problem
        self.computer = os.uname()
        self.computation_time = computation_time
        self._makespan = makespan
        self._seating_simulation = seating_simulation
        self.solver_description = solver_description
        self.solver_output = solver_output

    def __eq__(self, other):
        return self.problem == other.problem and self.assignment
    == other.assignment

    @property
    def seating_simulation(self):
        if self._seating_simulation:
            return self._seating_simulation
        else:
            self._seating_simulation = self.simulate_seating()
            return self._seating_simulation

    @property
    def makespan(self):
        if self._makespan:
```

```
            return self._makespan
        else:
            return self.seating_simulation.makespan

    def delay_passenger(self, passenger: int) -> "BapSolution":
        """
        :return: the same solution on an instance where the given passenger enters last
        """
        if passenger >= self.problem.num_passengers:
            raise ValueError(
                f"Passenger cannot be larger than {self.problem.num_passengers}"
            )

        new_problem = deepcopy(self.problem)

        p_stowing_speeds = new_problem.stowing_speeds[passenger]
        del new_problem.stowing_speeds[passenger]
        new_stowing_speeds = [p_stowing_speeds]
        new_stowing_speeds.extend(new_problem.stowing_speeds)
        new_problem.stowing_speeds = new_stowing_speeds

        p_walking_speeds = new_problem.walking_speeds[passenger]
        del new_problem.walking_speeds[passenger]
        new_walking_speeds = [p_walking_speeds]
        new_walking_speeds.extend(new_problem.walking_speeds)
        new_problem.walking_speeds = new_walking_speeds

        p_assignment = self.assignment[passenger]
        new_assignment = [p_assignment]
        new_assignment.extend(self.assignment)
        del new_assignment[passenger + 1]

        return BapSolution(new_problem, new_assignment)

    def swap_passengers(self, p_1: int, p_2: int) -> "BapSolution":
        """
        :return: the same solution on an instance where the two given passengers swap queueing positions
        """
        if any(p >= self.problem.num_passengers for p in (p_1, p_2)):
            raise ValueError(
                f"Passenger cannot be larger than {self.problem.num_passengers}"
            )

```

```
        new_problem = deepcopy(self.problem)

        p_1_stowing_speeds = new_problem.stowing_speeds[p_1]
        new_problem.stowing_speeds[p_1] = new_problem.stowing_speeds[p_2]
        new_problem.stowing_speeds[p_2] = p_1_stowing_speeds

        p_1_walking_speeds = new_problem.walking_speeds[p_1]
        new_problem.walking_speeds[p_1] = new_problem.walking_speeds[p_2]
        new_problem.walking_speeds[p_2] = p_1_walking_speeds

        new_assignment = deepcopy(self.assignment)
        p_1_pos = new_assignment[p_1]
        new_assignment[p_1] = new_assignment[p_2]
        new_assignment[p_2] = p_1_pos

        return BapSolution(new_problem, new_assignment)

    def change_speeds(self, passenger: int) -> "BapSolution":
        """
        :return: the same solution on an instance where speeds for the given passenger are changed
        """
        if passenger >= self.problem.num_passengers:
            raise ValueError(
                f"Passenger cannot be larger than {self.problem.num_passengers}"
            )

        new_problem = deepcopy(self.problem)
        new_times = AeroplaneBoardingProblem(
            num_passengers=1,
            num_rows=self.problem.num_rows,
            seats_per_row=self.problem.seats_per_row,
        )
        new_problem.walking_speeds[passenger], new_problem.stowing_speeds[passenger] = (
            new_times.walking_speeds[0],
            new_times.stowing_speeds[0],
        )

        return BapSolution(new_problem, self.assignment)

    def combined_disturbance(self) -> "BapSolution":
        """
        :return: the same solution on an instance where all available disturbances have been applied once
        """
```

```
        p = [random.randint(0, self.problem.num_passengers - 1) for _ in range(4)]
        return (
            self.delay_passenger(p[0]).swap_passengers(p[1], p[2]).change_speeds(p[3])
        )

    def simulate_seating(self) -> SeatingSimulation:
        """
        Computes a seating simulation for this aeroplane boarding solution.
        This includes computation of the makespan.
        """
        bap = self.problem
        passenger_seated_times = [0 for _ in range(bap.num_passengers)]
        passenger_enters_row = []
        row_blockage = [0 for _ in range(bap.num_rows)]

        for passenger in range(bap.num_passengers):
            assigned_row = self.assignment[passenger]
            passenger_enters_row.append([0 for _ in range(assigned_row + 1)])

            for row in range(assigned_row + 1):
                passenger_enters_row[passenger][row] = (
                    row_blockage[0]
                    if row == 0
                    else max(
                        passenger_enters_row[passenger][row - 1]
                        + bap.walking_speeds[passenger][row - 1],
                        row_blockage[row],
                    )
                )

                if not row == 0:
                    row_blockage[row - 1] = passenger_enters_row[passenger][row]

                if row == assigned_row:
                    passenger_seated_time = (
                        passenger_enters_row[passenger][row]
                        + bap.stowing_speeds[passenger][row]
                    )

                    row_blockage[row] = passenger_seated_time
                    passenger_seated_times[passenger] = passenger_seated_time

```

```
        makespan = max(passenger_seated_times, default=0)
        seating_simulation = SeatingSimulation(
            passenger_seated_times, passenger_enters_row, makespan, solution=self
        )
        return seating_simulation


class SeatAssigner(ABC):
    """
    Abstract base class for SeatAssigner objects. Used to define a common interface.
    """

    def solve(self, bap: "AeroplaneBoardingProblem", **kwargs) -> BapSolution:
        """
        Calls the solve implementation, adds timing information and the solver description.
        """
        start_time = datetime.now()
        solution = self.solve_implementation(bap, **kwargs)
        solution.computation_time = datetime.now() - start_time
        solution.solver_description = type(self).__name__
        return solution

    def solve_implementation(self, bap: "AeroplaneBoardingProblem") -> BapSolution:
        """
        Returns a seat assignment for a given Aeroplane Boarding Problem and the makespan.
        """
        raise NotImplementedError


class AeroplaneBoardingProblem:
    """
    Any object of this class is an instance of the boarding an aeroplane problem.
    For a description of the problem, see the thesis.
    """

    @staticmethod
    def generate_common_bap_instance(rows: int, seats_per_row: int) -> dict:
        """
        Generates a dictionary encoding a benchmarking instance for both
        the seat assignment problem and the passenger reordering
```

```
problem in aeroplane boarding.
        """
        if any(v <= 0 for v in [rows, seats_per_row]):
            raise ValueError("All input parameters must be strictly positive.")

        passengers = seats_per_row * rows

        abp = AeroplaneBoardingProblem(
            num_rows=rows, num_passengers=passengers, seats_per_row=seats_per_row
        )

        walking_speeds = abp.walking_speeds
        stowing_speeds = abp.stowing_speeds

        seat_assignment = [
            (r, s) for r, s in product(range(1, rows + 1), range(1, seats_per_row + 1))
        ]
        random.shuffle(seat_assignment)

        instance = {
            "rows": rows,
            "seats_per_row": seats_per_row,
            "walking_speeds": walking_speeds,
            "stowing_speeds": stowing_speeds,
            "seat_assignment": seat_assignment,
        }
        return instance

    @staticmethod
    def write_common_instance_to_disk(instance: dict, file_name):
        def stringify_passenger(p: int):
            seat = instance["seat_assignment"][p]
            settle_times = " ".join(str(n) for n in instance["stowing_speeds"][p])
            travel_times = " ".join(str(n) for n in instance["walking_speeds"][p])
            return f"row {seat[0]}\ncolumn {seat[1]}\nsettle_times {settle_times}\ntravel_times {travel_times}"

        rows = instance["rows"]
        seats_per_row = instance["seats_per_row"]
        passengers = rows * seats_per_row
        s = f"n_rows {rows}\nn_columns {seats_per_row}\nn_passengers {passengers}\n"
        s += "\n".join(stringify_passenger(p) for p in range(passengers))
```

```
        s += "\n"

        with open(file_name, "w") as f:
            f.write(s)

    @classmethod
    def load_common_instance_from_disk(cls, file_name) -> "
AeroplaneBoardingProblem":
        with open(file_name, "r") as f:
            lines = f.readlines()

        # parse size parameters
        for par_line in lines[:3]:
            if par_line.startswith("n_rows"):
                rows = int(par_line.split()[-1])
            elif par_line.startswith("n_columns"):
                seats_per_row = int(par_line.split()[-1])
            elif par_line.startswith("n_passengers"):
                passengers = int(par_line.split()[-1])

        # parse passenger parameters
        walking_speeds = [None for _ in range(passengers)]
        stowing_speeds = [None for _ in range(passengers)]

        for p in range(passengers):
            s = 3 + p * 4
            e = s + 4
            for par_line in lines[s:e]:
                if par_line.startswith("settle"):
                    stowing_speeds[p] = [int(float(i)) for i in
par_line.split()[1:]]
                elif par_line.startswith("travel"):
                    walking_speeds[p] = [int(float(i)) for i in
par_line.split()[1:]]

        return cls(
            num_rows=rows,
            num_passengers=passengers,
            seats_per_row=seats_per_row,
            stowing_speeds=stowing_speeds,
            walking_speeds=walking_speeds,
        )

    @staticmethod
    def write_to_disk(problems: List["AeroplaneBoardingProblem"],
file_name):
        with open(file_name, "wb") as file:
            pickle.Pickler(file).dump(problems)

```

```
    @staticmethod
    def load_from_disk(file_name) -> List["AeroplaneBoardingProblem"]:
        try:
            with open(file_name, "rb") as file:
                return pickle.Unpickler(file).load()
        except UnpicklingError as ue:
            return [AeroplaneBoardingProblem.load_common_instance_from_disk(file_name)]

    def __init__(
        self,
        num_rows: int = None,
        num_passengers: int = None,
        seats_per_row: int = None,
        stowing_speeds: List[List[int]] = None,
        walking_speeds: List[List[int]] = None,
        classes: int = None,
    ):
        """
        When arguments are omitted, this initializer acts as a generator for a randomised instance.
        The walking and stowing speed lists is indexed by passenger and by row, in that order, both starting at 0.
        If a number of classes is specified,
        all passengers are from a pool with that number of different passenger types.
        """
        self.seats_per_row = (
            seats_per_row if seats_per_row is not None else random.randint(1, 7)
        )
        self.num_passengers = (
            num_passengers if num_passengers is not None else random.randint(0, 50)
        )
        required_num_rows = math.ceil(self.num_passengers / self.seats_per_row)
        self.num_rows = (
            num_rows
            if num_rows is not None
            else random.randint(required_num_rows, required_num_rows + 10)
        )

        if self.num_rows * self.seats_per_row == 0:
            raise ValueError("Plane does not have any seats.")

        if self.num_rows * self.seats_per_row < self.
```

```
num_passengers:
            raise ValueError("Plane does not have enough seats for all passengers.")

        num_templates = classes if classes else self.num_passengers

        def generate_walking_value():
            return random.choices([1, 2, 3], weights=[1, 2, 1])[0]

        def generate_stowing_value():
            z = np_rand.normal(60, 20)
            return max(min(int(z), 120), 1)

        def generate_template(generator):
            return [
                [generator() for _ in range(self.num_rows)]
                for _ in range(num_templates)
            ]

        if classes and (walking_speeds or stowing_speeds):
            raise ValueError(
                "Using classes with custom defined speeds is not supported at the moment."
            )

        stowing_templates = generate_template(generate_stowing_value)
        walking_templates = generate_template(generate_walking_value)

        if classes:
            # draw from templates and initialise properly
            self.stowing_speeds: List[List[int]] = []
            self.walking_speeds: List[List[int]] = []
            for passenger in range(self.num_passengers):
                template_index = random.randint(0, classes - 1)
                self.stowing_speeds.append(stowing_templates[template_index])
                self.walking_speeds.append(walking_templates[template_index])

        else:
            self.stowing_speeds = (
                stowing_speeds if stowing_speeds else stowing_templates
            )
            self.walking_speeds = (
                walking_speeds if walking_speeds else
```

```
walking_templates
            )

        if len(self.stowing_speeds) != self.num_passengers:
            raise ValueError(
                f"The length of the stowing speed list ({len(self.stowing_speeds)}) does not match the number of passengers ({self.num_passengers})."
            )

        if len(self.walking_speeds) != self.num_passengers:
            raise ValueError(
                f"The length of the walking speed list ({len(self.walking_speeds)}) does not match the number of passengers ({self.num_passengers})."
            )

        # extend walking speed lists if needed
        for l in self.walking_speeds:
            if len(l) > self.num_rows:
                raise ValueError(
                    f"At least one walking speed list is too long, the maximum length is {self.num_rows}"
                )
            else:
                try:
                    filler = l[-1]
                except IndexError:
                    filler = generate_walking_value()
                l.extend(repeat(filler, self.num_rows - len(l)))

        # extend stowing speed lists if needed
        for l in self.stowing_speeds:
            if len(l) > self.num_rows:
                raise ValueError(
                    f"At least one stowing speed list is too long, the maximum length is {self.num_rows}"
                )
            else:
                try:
                    filler = l[-1]
                except IndexError:
                    filler = generate_stowing_value()
                l.extend(repeat(filler, self.num_rows - len(l)))

        self.solutions: List[BapSolution] = list()
        self.classes = classes

    def __str__(self):
```

```
        s = "Aeroplane Boarding Problem"
        s += f"classes: {self.classes if self.classes else 'freely chosen'}"
        s += f"\npassengers: {self.num_passengers}\nrows: {self.num_rows}\nseats per row: {self.seats_per_row}\nwalking and stowing speeds: ---------"
        for p in self.passengers:
            s += f"\npassenger {p}:\nwalking: {self.walking_speeds[p]}\nstowing: {self.stowing_speeds[p]}"
        return s

    @property
    def passengers(self):
        return range(self.num_passengers)

    @property
    def rows(self):
        return range(self.num_rows)

    def compute_makespan(self, assignment: Dict) -> float:
        """
        Computes the makespan for a given seat assignment.
        """
        raise NotImplementedError

    def solve(self, assigner: SeatAssigner, **kwargs):
        """
        Compute a seat assignment using the given assigner.
        """
        solution = assigner.solve(bap=self, **kwargs)
        self.solutions.append(solution)
        return solution


class RandomAssigner(SeatAssigner):
    """
    Assigns seats at random
    """

    def solve_implementation(self, bap: AeroplaneBoardingProblem) -> BapSolution:
        row_tickets = list(
            chain.from_iterable(repeat(r, bap.seats_per_row) for r in bap.rows)
        )
        random.shuffle(row_tickets)
        assignment = row_tickets[: bap.num_passengers]
        return BapSolution(assignment=assignment, problem=bap)

```

```python

class DirectionalAssigner(SeatAssigner):
    """
    Assigns seats in one direction, either back to front or front to back.
    """

    def __init__(self, reverse: bool):
        self.reverse = reverse

    def solve_implementation(self, bap: AeroplaneBoardingProblem) -> BapSolution:
        assignment = [0 for _ in bap.passengers]
        usable_rows = int(
            math.ceil(float(bap.num_passengers) / float(bap.seats_per_row))
        )
        front_rows = list(bap.rows)[:usable_rows]
        row_iterator = reversed(front_rows) if self.reverse else front_rows
        for passenger, row in zip(
            bap.passengers,
            chain.from_iterable(repeat(row, bap.seats_per_row) for row in row_iterator),
        ):
            assignment[passenger] = row
        return BapSolution(assignment=assignment, problem=bap)


class FrontToBackAssigner(DirectionalAssigner):
    """
    Assigns seats front to back. Can be used as a starting solution for an improvement heuristics.
    """

    def __init__(self):
        super().__init__(reverse=False)


class BackToFrontAssigner(DirectionalAssigner):
    """
    Assigns seats back to front. Useful as starting point for improvement heuristics.
    """

    def __init__(self):
        super().__init__(reverse=True)


```

```python
class WindowToWindowAssigner(SeatAssigner):
    """
    Assigns seats window to window
    """

    def solve_implementation(self, bap: AeroplaneBoardingProblem) -> BapSolution:
        btf = list(reversed(list(bap.rows)))
        assignment = list(chain.from_iterable(repeat(btf, bap.seats_per_row)))[
            : bap.num_passengers
        ]
        return BapSolution(bap, assignment)


class ReversePyramidAssigner(SeatAssigner):
    """
    Assigns seats in a reverse pyramid scheme
    """

    def solve_implementation(self, bap: AeroplaneBoardingProblem) -> BapSolution:
        if not bap.seats_per_row % 2 == 0:
            raise ValueError(
                f"Layout has {bap.seats_per_row} seats per row, which is not even, as is required for the reverse pyramid assigner."
            )

        # only assign to half of the columns and copy assignment
        first_row_in_column = {c: 0 for c in range(int(bap.seats_per_row / 2))}

        def push_into_column(c: int) -> Optional[int]:
            """
            :param c: The column to push into
            :return: The row pushed into if successful, None otherwise
            """
            first_row = first_row_in_column[c]
            if first_row >= bap.num_rows:
                return None
            else:
                first_row_in_column[c] += 1
                return first_row

        def push_into_lowest() -> Tuple[int, int]:
            """
            :return: the row and column pushed into
```

```
            """
            column = 0
            row = push_into_column(column)
            while row is None:
                column += 1
                row = push_into_column(column)
            return row, column

        num_groups = 6
        split = 0.6
        passengers = list(bap.passengers)[: int(math.ceil(bap.num_passengers / 2))]
        group_size = max(int(math.floor(len(passengers) / num_groups)), 1)

        passenger_groups = []
        for i in range(num_groups - 1):
            passenger_groups.append(passengers[i * group_size : (i + 1) * group_size])
        passenger_groups.append(passengers[group_size * (num_groups - 1) :])

        assignment = dict()

        # handle first group
        for p in passenger_groups[0]:
            assignment[p] = push_into_lowest()

        # handle other groups
        last_column = 0
        for group in passenger_groups[1:]:
            # push first chunk into lowest
            split_index = int(math.floor(len(group) * split))
            for p in group[:split_index]:
                assignment[p] = push_into_lowest()
                _, last_column = assignment[p]

            # push second chunk higher
            if last_column < int(bap.seats_per_row / 2) - 1:
                last_column += 1

            for p in group[split_index:]:
                row = push_into_column(last_column)
                while row is None:
                    last_column += 1
                    row = push_into_column(last_column)
                assignment[p] = row, last_column

        row_assignment = list(
```

```
                chain.from_iterable(
                    [
                        bap.num_rows - 1 - assignment[p][0],
                        bap.num_rows - 1 - assignment[p][0],
                    ]
                    for p in passengers
                )
            )[: bap.num_passengers]

        return BapSolution(bap, row_assignment)


class SteffenMethodAssigner(SeatAssigner):
    """
    Implements the method invented by Jason H. Steffen
    """

    def solve_implementation(self, bap: AeroplaneBoardingProblem)
 -> BapSolution:
        if not bap.seats_per_row % 2 == 0:
            raise ValueError(
                f"Layout has {bap.seats_per_row} seats per row,
 which is not even, as is required for the Steffen's Method
 assigner."
            )

        # generate column filling pattern
        interleaved = zip(range(bap.seats_per_row), reversed(range
 (bap.seats_per_row)))
        interleaved_doubled = chain.from_iterable(repeat(i, 2) for
  i in interleaved)
        groups_with_offset = zip(interleaved_doubled, cycle([False
 , True]))
        columns_with_offset = (
            ((t[0], offset), (t[1], offset)) for t, offset in
 groups_with_offset
        )
        column_sequence = islice(
            chain.from_iterable(columns_with_offset), 2 * bap.
 seats_per_row
        )

        assignment = list()

        for column, use_offset in column_sequence:
            initial_row = bap.num_rows - 2 if use_offset else bap.
 num_rows - 1
            for row in range(initial_row, -1, -2):
                assignment.append(row)
```

```python

        return BapSolution(bap, assignment[: bap.num_passengers])


class LocalSearchAssigner(SeatAssigner):
    """
    Improve an initial seat assignment using local search.
    """

    def __init__(
        self,
        initializer: Optional[SeatAssigner] = None,
        initial_solution: Optional[BapSolution] = None,
        **kwargs,
    ):
        self.initializer = initializer
        self.initial_solution = initial_solution

    def solve_implementation(self, bap: AeroplaneBoardingProblem)
        -> BapSolution:
        if self.initial_solution is not None:
            current_solution = self.initial_solution
        else:
            current_solution = self.initializer.solve(bap)
        best_makespan = current_solution.makespan
        improvement_possible = True

        empty_seats_per_row = {
            r: bap.seats_per_row
            - sum(1 for p in bap.passengers if current_solution.
    assignment[p] == r)
            for r in bap.rows
        }

        def swap(p1: int, p2: int):
            current_solution._makespan = None
            current_solution._seating_simulation = None
            old_p1 = current_solution.assignment[p1]
            current_solution.assignment[p1] = current_solution.
    assignment[p2]
            current_solution.assignment[p2] = old_p1

        def swap_into_empty_seat(p: int, target_row: int) -> int:
            """
            Returns the row that was swapped out of.
            """
            if not empty_seats_per_row[target_row] > 0:
                raise ValueError(f"There is no empty seat in row {
    target_row}")
```

```

            old_row = current_solution.assignment[p]
            current_solution.assignment[p] = target_row

            empty_seats_per_row[old_row] += 1
            empty_seats_per_row[target_row] -= 1

            return old_row

        while improvement_possible:
            improvement_possible = False

            passengers = list(bap.passengers)
            random.shuffle(passengers)

            for passenger in passengers:
                other_passengers = list(range(passenger))
                random.shuffle(other_passengers)

                # swap into empty seats
                empty_seats = [r for r in bap.rows if empty_seats_per_row[r] > 0]
                random.shuffle(empty_seats)

                for target_row in empty_seats:
                    old_row = swap_into_empty_seat(passenger, target_row)
                    if current_solution.makespan < best_makespan:
                        best_makespan = current_solution.makespan
                        improvement_possible = True
                        break
                    else:  # swap back
                        swap_into_empty_seat(passenger, old_row)

                # swap among passengers
                for other_passenger in other_passengers:
                    swap(passenger, other_passenger)
                    if current_solution.makespan < best_makespan:
                        best_makespan = current_solution.makespan
                        improvement_possible = True
                    else:  # swap back
                        swap(passenger, other_passenger)

        current_solution._makespan = None
        current_solution._seating_simulation = None
        return current_solution


class MultiSearchAssigner(SeatAssigner):
```

```
    """
    Uses local search to find solution. Has multiple starting points.
    """

    def __init__(self, tries: int):
        self.start_assigners: List[SeatAssigner] = [
            RandomAssigner() for _ in range(tries)
        ]

    def solve_implementation(self, bap: AeroplaneBoardingProblem) -> BapSolution:
        solutions = [
            LocalSearchAssigner(assigner).solve_implementation(bap)
            for assigner in self.start_assigners
        ]

        return min(solutions, key=lambda s: s.makespan)


class MIPExactSeatAssigner(SeatAssigner):
    """
    Exactly solves the Aeroplane Boarding Problem using gurobi and
    mixed integer programming.
    """

    def __init__(
        self,
        start_solution: Optional[BapSolution] = None,
        use_additional_cuts: bool = False,
        **kwargs,
    ):
        self.solution_random_id = random.randint(0, 1_000_000)
        self.start_solution = start_solution
        self.use_additional_cuts = use_additional_cuts

    def get_iis_for_solution(
        self, bap: AeroplaneBoardingProblem, sol: BapSolution
    ) -> str:
        """
        Tests a solution for feasibility and returns the infeasible subsystem. Fails otherwise.
        """
        model, pass_in_row, _, _, _ = self.get_gurobi_model(bap)
        for p in bap.passengers:
            model.addConstr(pass_in_row[p, sol.assignment[p]] == 1)
        model.optimize()
```

```python

        with NamedTemporaryFile(suffix=".ilp", mode="w") as f:
            model.computeIIS()
            model.write(f.name)
            iis_str = f.read()
        return iis_str.decode()

    def get_gurobi_model(
        self, bap: AeroplaneBoardingProblem
    ) -> Tuple[gurobipy.Model, dict, dict, gurobipy.Var]:
        """
        Returns a gurobi model encoding of the aeroplane boarding problem.
        Also returns a dictionary with the relevant decision variables and the makespan variable.
        """
        model = gurobipy.Model(f"MIP generated from aeroplane boarding problem")
        model.setAttr("ModelSense", gurobipy.GRB.MINIMIZE)
        model.message(f"MODEL_ID {self.solution_random_id}")

        pass_in_row = {
            (passenger, row): model.addVar(
                vtype=gurobipy.GRB.BINARY, name=f"p{passenger}_in_r{row}"
            )
            for passenger, row in product(bap.passengers, bap.rows)
        }

        pass_enters_row = {
            (passenger, row): model.addVar(
                vtype=gurobipy.GRB.CONTINUOUS,
                name=f"p{passenger}_enters_r{row}",
                lb=0.0,
            )
            for passenger, row in product(bap.passengers, range(bap.num_rows + 1))
        }

        M = max(
            (
                max(bap.walking_speeds[p][r], bap.stowing_speeds[p][r])
                for p, r in product(bap.passengers, bap.rows)
            ),
            default=0,
        ) * bap.num_rows

```

```
        makespan = model.addVar(
            vtype=gurobipy.GRB.CONTINUOUS, lb=0.0, obj=1.0, name="makespan"
        )

        # no row exceeds capacity
        for row in bap.rows:
            model.addConstr(
                quicksum(pass_in_row[p, row] for p in bap.passengers)
                <= bap.seats_per_row
            )

        for passenger in bap.passengers:
            # every passenger has a seat
            model.addConstr(
                quicksum(pass_in_row[passenger, row] for row in bap.rows) == 1
            )
            # makespan conditions
            model.addConstr(makespan >= pass_enters_row[passenger, bap.num_rows])

            for row in bap.rows:
                # respect moving and stowing times
                model.addConstr(
                    pass_enters_row[passenger, row + 1]
                    >= pass_enters_row[passenger, row]
                    + bap.stowing_speeds[passenger][row] * pass_in_row[passenger, row]
                    + bap.walking_speeds[passenger][row]
                    * quicksum(
                        pass_in_row[passenger, r] for r in range(row + 1, bap.num_rows)
                    )
                )

                if self.use_additional_cuts:
                    # add lower bounds on arrival times in rows
                    model.addConstr(
                        pass_enters_row[passenger, row + 1]
                        >= (
                            quicksum(
                                bap.walking_speeds[passenger][r] for r in range(row)
                            )
                            + bap.stowing_speeds[passenger][row]
                        )
                        * pass_in_row[passenger, row]
```

```
                    )

                # only enter row once all others have left
                for other_passenger in range(passenger):
                    model.addConstr(
                        pass_enters_row[passenger, row]
                        >= pass_enters_row[other_passenger, row +
 1]
                        - M
                        * (
                            1
                            - quicksum(
                                pass_in_row[passenger, r]
                                for r in range(row, bap.num_rows)
                            )
                        )
                    )

        model.setParam("TimeLimit", 2 * 60 * 60)

        return model, pass_in_row, pass_enters_row, makespan

    def set_initial_solution(
        self,
        bap: AeroplaneBoardingProblem,
        pass_in_row: Dict[Tuple[int, int], gurobipy.Var],
    ):
        initial_solution = self.start_solution
        for p, r in product(bap.passengers, bap.rows):
            pass_in_row[p, r].Start = 1 if initial_solution.
 assignment[p] == r else 0

    def get_relaxed_solution(self, bap: AeroplaneBoardingProblem):
        """
        Computes the solution of the relaxed model and returns it.
        """
        model, pass_in_row, pass_enters_row, _ = self.
 get_gurobi_model(bap)

        pir = dict()
        per = dict()

        def callback(model, where):
            if where == gurobipy.GRB.Callback.MIPNODE:
                for p, r in product(bap.passengers, bap.rows):
                    pir[p, r] = model.cbGetNodeRel(pass_in_row[(p,
  r)])
                    per[p, r] = model.cbGetNodeRel(pass_enters_row
 [(p, r)])
```

```python

        model.setParam("NodeLimit", 1)
        model.optimize(callback)

        return pir, per

    def show_relaxed_solution(
        self, bap: AeroplaneBoardingProblem, file_name=None, random_seed=None
    ):
        """
        Shows a timing graph for the relaxed root node solution.
        """

        pass_in_row, pass_enters_row = self.get_relaxed_solution(bap)

        fig = plt.figure()
        ax = fig.add_subplot(111)
        handles = []

        if random_seed:
            random.seed(random_seed)

        for p in bap.passengers:

            def random_colour():
                return random.random(), random.random(), random.random()

            colour = random_colour()
            while not sum(colour) >= 1:
                colour = random_colour()

            for r in bap.rows:
                width = sum(pass_in_row[p, row] for row in range(r, bap.num_rows))
                height = pass_enters_row[p, r + 1] - pass_enters_row[p, r]
                rect = Rectangle(
                    (r, pass_enters_row[p, r]),
                    width,
                    height,
                    fc=colour,
                    alpha=0.5,
                    label=f"passenger {p}",
                )
                ax.add_patch(rect)
                if r == 0:
```

```
                    handles.append(rect)

        ax.set_xscale("linear")
        ax.set_xlabel("rows")
        ax.set_yscale("linear")
        ax.set_ylabel("time")

        ax.set_title(f"Fractional solution for {bap.num_passengers} passengers")

        # plt.legend(handles=handles)

        if file_name:
            with open(file_name, "w") as f:
                for p in bap.passengers:
                    f.write(f"passenger {p}:\n")
                    for r in bap.rows:
                        f.write(
                            f"row {r}:\t{pass_in_row[p, r]}\t {pass_enters_row[p, r]}\n"
                        )

        plt.show()

    def solve_implementation(
        self, bap: AeroplaneBoardingProblem, **kwargs
    ) -> BapSolution:
        """
        Returns an optimal assignment of passengers to seats as a dictionary and the optimal objective value.
        """
        model, pass_in_row, pass_enters_row, makespan = self.get_gurobi_model(bap)
        if self.start_solution:
            self.set_initial_solution(bap, pass_in_row)
        model.optimize()

        if model.Status == gurobipy.GRB.INFEASIBLE:
            model.computeIIS()
            model.write("out.ilp")
            print("Walking speeds: ")
            print(bap.walking_speeds)
            print("Stowing speeds:")
            print(bap.stowing_speeds)

        assignment = []
        for p in bap.passengers:
            for r in bap.rows:
                if pass_in_row[p, r].X > 0.5:
```

```
                    assignment.append(r)

        return BapSolution(assignment=assignment, makespan=makespan.X, problem=bap)


class AlternativeMIPAssigner(MIPExactSeatAssigner):
    """
    Uses an alternative MIP formulation
    """

    def get_iis_for_solution(
        self, bap: AeroplaneBoardingProblem, sol: BapSolution
    ) -> str:
        model, pass_in_seat, _, _, _ = self.get_gurobi_model(bap)
        next_assignable_seat = [0 for _ in bap.rows]
        for p in bap.passengers:
            r = sol.assignment[p]
            s = next_assignable_seat[r]
            next_assignable_seat[r] += 1
            model.addConstr(pass_in_seat[p, r, s] == 1)
        model.optimize()

        with NamedTemporaryFile(suffix=".ilp") as f:
            model.computeIIS()
            model.write(f.name)
            iis_str = f.read()
        return iis_str.decode()

    def set_initial_solution(
        self,
        bap: AeroplaneBoardingProblem,
        pass_in_seat: Dict[Tuple[int, int, int], gurobipy.Var],
        seat_arrival_times: Dict[Tuple[int, int, int], gurobipy.Var],
    ):
        initial_solution = self.start_solution
        first_available_seat = {r: 0 for r in bap.rows}
        passenger_seated = {p: False for p in bap.passengers}

        for p, r, s in product(bap.passengers, bap.rows, range(bap.seats_per_row)):
            start_value = 0
            if initial_solution.assignment[p] == r:
                if first_available_seat[r] == s and not passenger_seated[p]:
                    first_available_seat[r] += 1
                    passenger_seated[p] = True
                    start_value = 1
```

```python

                    # Copy seat arrival times
                    for row in range(r + 1):
                        seat_arrival_times[
                            (r, s, row)
                        ].Start = initial_solution.seating_simulation.passenger_enters_row[
                            p
                        ][
                            row
                        ]

            pass_in_seat[p, r, s].Start = start_value

    def solve_implementation(
        self, bap: AeroplaneBoardingProblem, **kwargs
    ) -> BapSolution:
        """
        Returns an optimal assignment of passengers to seats as a dictionary and the optimal objective value.
        """
        model, pass_in_seat, seat_arrival_times, seat_departure_times, M, makespan = self.get_gurobi_model(
            bap
        )

        if self.start_solution:
            self.set_initial_solution(bap, pass_in_seat, seat_arrival_times)

        model.setParam("LazyConstraints", 1)
        model.optimize()

        assignment = []
        for p in bap.passengers:
            for r in bap.rows:
                if any(
                    pass_in_seat[(p, r, s)].X > 0.5 for s in range(bap.seats_per_row)
                ):
                    assignment.append(r)

        return BapSolution(assignment=assignment, makespan=makespan.X, problem=bap)

    def get_gurobi_model(self, bap: AeroplaneBoardingProblem):
        """
        Implements an alternative MIP formulation of the problem.
        """
```

```
        env = gurobipy.Env(GUROBI_LOG_NAME)
        model = gurobipy.Model(
            f"Alternative MIP generated from aeroplane boarding problem", env
        )
        model.setParam("Heuristics", 0)
        model.setAttr("ModelSense", gurobipy.GRB.MINIMIZE)
        model.message(f"MODEL_ID {self.solution_random_id}")

        M = (
            max(
                max(bap.walking_speeds[p][r], bap.stowing_speeds[p][r])
                for p, r in product(bap.passengers, bap.rows)
            )
            * bap.num_rows
        )

        pass_in_seat = {
            (passenger, row, seat): model.addVar(
                vtype=gurobipy.GRB.BINARY, name=f"p{passenger}_in_s{seat}_in_r{row}"
            )
            for passenger, row, seat in product(
                bap.passengers, bap.rows, range(bap.seats_per_row)
            )
        }

        seat_arrival_times = {
            (row, seat, r): model.addVar(
                vtype=gurobipy.GRB.INTEGER, name=f"arr_r{row}_s{seat}_in_r{r}"
            )
            for row, seat in product(bap.rows, range(bap.seats_per_row))
            for r in range(row + 1)
        }

        seat_departure_times = {
            (row, seat, r): model.addVar(
                vtype=gurobipy.GRB.INTEGER, name=f"dep_r{row}_s{seat}_from_r_{r}"
            )
            for row, seat in product(bap.rows, range(bap.seats_per_row))
            for r in range(row + 1)
        }

        makespan = model.addVar(vtype=gurobipy.GRB.INTEGER, obj=1,
```

```
 name="makespan")

        # Makespan conditions
        for row, seat in product(bap.rows, range(bap.seats_per_row)):
            model.addConstr(makespan >= seat_departure_times[(row, seat, row)])

        # Every passenger has exactly one seat
        for passenger in bap.passengers:
            model.addConstr(
                quicksum(
                    pass_in_seat[(passenger, row, seat)]
                    for row, seat in product(bap.rows, range(bap.seats_per_row))
                )
                == 1
            )

        # At most one passenger is in every seat
        for row, seat in product(bap.rows, range(bap.seats_per_row)):
            model.addConstr(
                quicksum(
                    pass_in_seat[(passenger, row, seat)] for passenger in bap.passengers
                )
                <= 1
            )

        for row, seat in product(bap.rows, range(bap.seats_per_row)):
            # couple arrive and leave times
            for r in range(1, row + 1):
                model.addConstr(
                    seat_arrival_times[(row, seat, r)]
                    == seat_departure_times[(row, seat, r - 1)]
                )

            # respect moving times
            for r in range(row):
                model.addConstr(
                    seat_departure_times[(row, seat, r)]
                    >= seat_arrival_times[(row, seat, r)]
                    + quicksum(
                        bap.walking_speeds[p][r] * pass_in_seat[(p, row, seat)]
                        for p in bap.passengers
                    )
```

```python
                )

            # respect stowing times
            model.addConstr(
                seat_departure_times[(row, seat, row)]
                == seat_arrival_times[(row, seat, row)]
                + quicksum(
                    bap.stowing_speeds[p][row] * pass_in_seat[(p, row, seat)]
                    for p in bap.passengers
                )
            )

        # break symmetries
        for p, r in product(bap.passengers, bap.rows):
            for right_seat in range(bap.seats_per_row):
                for left_seat in range(right_seat):
                    model.addConstr(
                        1 - pass_in_seat[p, r, right_seat]
                        >= quicksum(
                            pass_in_seat[other_p, r, left_seat]
                            for other_p in range(p, bap.num_passengers)
                        )
                    )

        # transfer passenger order
        for r, s in product(bap.rows, range(bap.seats_per_row)):
            for o_r, o_s in product(bap.rows, range(bap.seats_per_row)):
                if (r, s) != (o_r, o_s):
                    for row in range(min(r, o_r) + 1):
                        for p in bap.passengers:
                            c = model.addConstr(
                                seat_arrival_times[r, s, row]
                                >= seat_departure_times[o_r, o_s, row]
                                - M
                                * (
                                    2
                                    - pass_in_seat[p, r, s]
                                    - quicksum(
                                        pass_in_seat[o_p, o_r, o_s] for o_p in range(p)
                                    )
                                )
                            )
                            c.setAttr("Lazy", 1)

```

```
        model.setParam("TimeLimit", 2 * 60 * 60)

        return (
            model,
            pass_in_seat,
            seat_arrival_times,
            seat_departure_times,
            M,
            makespan,
        )
```

A.2 Instance Generation

```
"""
Contains the generation code for instances where times vary per row
"""

import os.path as op

from lib import AeroplaneBoardingProblem

NUM_INSTANCES = 10
CONFIGURATIONS = [(10, 2), (20, 2), (20, 4), (30, 6)]

def generate_instances():
    for c in CONFIGURATIONS:
        rows, seats_per_row = c
        for i in range(NUM_INSTANCES):
            instance = AeroplaneBoardingProblem.generate_common_bap_instance(rows, seats_per_row)
            file_name = f"own_{rows}_{seats_per_row}_{i}.abp"
            file_name = op.abspath(op.join("instances/own", file_name))
            AeroplaneBoardingProblem.write_common_instance_to_disk(instance, file_name)


if __name__ == "__main__":
    generate_instances()
```

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

___________________________ ___________________________

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

____

____

____

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

___________________________ ___________________________

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

I have read and understood the above official notification:

___________________________ ___________________________

Ort, Datum/City, Date

Unterschrift/Signature